Département d'Informatique
ENS de Cachan

Petit manuel de survie pour C++

François Laroussinie
fl@lsv.ens-cachan.fr

Année 2004-2005

Petit manuel de survie pour C++

à compléter avec les précieux commentaires faits en cours...

François Laroussinie

fl@lsv.ens-cachan.fr

version du 2 février 2005

A. Structures de base de C++

1 Introduction

Ces notes ne sont qu'une rapide introduction aux aspects élémentaires du langage C++, elles doivent vous permettre d'écrire des programmes simples et de vous familiariser avec le langage. L'héritage ou les types génériques ne sont pas abordés. Ces notes ne remplacent pas des ouvrages complets [1, 2] ; [3] est un bon manuel de référence ; [4] est intéressant mais daté. Les * (resp. **) signalent des notions qui peuvent être laissées (resp. qu'il faut laisser) de côté dans un premier temps. Il est conseillé de bien maîtriser la partie A avant d'aborder les parties B et C (la partie C est indépendante de la partie B, sauf les sections 19 et 20).
La lecture de ces notes doit être accompagnée par une pratique intensive du C++... A vos claviers !

Une fois écrit dans un fichier, par exemple `toto.cc`, un programme [1] C++ doit être compilé par la commande : `g++ -o toto toto.cc`
Cela produit un fichier exécutable `toto` correspondant au programme. Pour l'exécuter, il suffit de taper : `./toto`
En cas de question : `fl@lsv.ens-cachan.fr`

2 Variables

Les types élémentaires sont :
- `int` : entiers (au min. codés sur 16bits, permettant de stocker des valeurs ≤ 32767) (d'autres formats existent : `long int` (min. 32 bits) , `short int` ou `unsigned int`),
- `double` (ou `float`) : nombres "réels" (23.3 ou $2.456e12$ ou $23.56e-4$),
- `char` : caractères ('a','b',... 'A',... ,':',...),
- `bool` : booléens ('true' ou 'false').

Les définitions de variables se font la manière suivante :
Syntaxe : *type v*;
Cette instruction définit la variable v de type *type* et peut apparaître n'importe où dans un

[1] Les impatients trouveront à la section 10.5 la structure d'un programme C++ et un court exemple de programme prêt à compiler...

programme (mais avant la première utilisation de la variable !). Cette variable existe jusqu'à la fin de la première instruction composée (marquée par }) qui contient cette définition.

Une variable définie en dehors de toute fonction et de tout espace de nom [2] est une *variable globale* : elle peut être utilisée dans toutes les fonctions à partir de sa définition. L'utilisation de variables globales pose des problèmes de lisibilité et donc de sûreté, il ne faut pas en abuser : On peut (pratiquement) toujours se dispenser d'en utiliser ...

Une variable peut être *initialisée* lors de sa déclaration, deux notations sont possibles :

```
int p=34;
double x=12.34;
```

ou

```
int p (34);
double x (12.34);
```

Une variable d'un type élémentaire non initialisée n'a pas de valeur définie, i.e. peut contenir n'importe quoi.

3 Constantes symboliques

Il est possible de définir des constantes symboliques avec le mot clé `const` :
Syntaxe : `const` *type nom* = *val*;
Par exemple : `const int Taille = 100 ;`
Il ne sera pas possible de modifier `Taille` dans le reste du programme (erreur à la compilation).

4 Chaînes de caractères et tableaux

Il existe plusieurs possibilités pour définir des tableaux et des chaînes de caractères, nous allons d'abord présenter (succinctement) les classes `vector` et `string`. Dans la section 17, nous présenterons la méthode classique ("à la C") pour ces deux structures.

4.1 Chaînes de caractères - `string`

Pour utiliser la classe `string`, il faut placer en tête du fichier : `# include <string>`

Ensuite, on peut définir des variables de chaînes de la manière habituelle. "`string t ;`" définit une variable de type `string`, "`string s(`N`,`c`) ;`" définit une variable `s` de longueur N (un entier) où chaque élément est initialisée avec le caractère c, "`string mot = "bonjour" ;`" définit une variable `mot` contenant la chaîne `bonjour`.

Etant donnée une chaîne `s`, `s.size()` représente la longueur de `s` et le i-ème caractère d'une chaîne `s` est désigné par `s[i]` avec `i` = 0,1,...`s.size()`-1. Le type exact de `s.size()` est `string::size_type` , cela correspond à un entier non signé pouvant contenir la taille de toute chaîne.

Etant données deux chaînes `s` et `t`, `s+t` est une nouvelle chaîne correspondant à la concaténation de `s` et `t`.

[2] Ce concept est expliqué dans la remarque 4 mais peut être ignoré en première lecture.

4.2 Tableaux - `vector`

Il existe deux types de tableau en C++ : d'une part il y a la classe `vector` et d'autre part, il y a les *tableaux simples* (les seuls disponibles dans le langage C) qui seront vus à la section 17.

Pour utiliser la classe `vector`, il faut placer en tête du fichier : `# include <vector>`

Un tableau est une structure typée : on distingue les tableaux sur les `int`, `char`, etc. On précise donc son type lors de la définition :

```
vector<int> Tab(100,5);   //  tableau d'entiers de taille 100 initialisé à 5
vector<int> Tab(50);      //  tableau d'entiers de taille 50 initialisé à 0
vector<double> T;         //  tableau de double, vide
```

La structure générale est donc "`vector<`*type*`>` Nom`(`n`,`v`) ;`" lorsqu'on initialise le tableau avec une valeur v. On peut aussi initialiser un tableau avec un autre tableau : "`vector<`*type*`>` $Nom1$ `=` $Nom2$ `;`", les valeurs de $Nom2$ sont alors recopiées dans $Nom1$.

Etant donné un tableau `T`, `T.size()` correspond à la taille de `T` et `T[i]` désigne le i-ème élément avec i = 0, ... `T.size()`-1. Comme pour les chaînes, `T.size()` renvoie un objet d'un type particulier, ici `vector<`*type*`>::size_type`, cela correspond à un entier non signé pouvant contenir la taille de tout `vector` de type *type*.

Les types `string` et `vector` comportent de nombreuses autres fonctions de manipulation, nous n'en donnons ici qu'un aperçu, nous reviendrons sur leurs caractéristiques dans le cours.

Remarque 1 *L'instruction "*`vector<vector<int> > T2;`*" définit un tableau à deux dimensions. Pour l'initialiser, on peut utiliser l'instruction suivante :*
`vector<vector<int> > T2(100,vector<int>(50,1));`
i.e. on initialise chacune des 100 cases de `T1` *avec un tableau de taille 50 rempli de 1 (*"`vector<int>(50,1)`*"). Le mystère de cette notation sera élucidé un peu plus tard. Patience...*

5 Commentaires

Pour commenter une (fin de) ligne : `// suivi de ce que l'on veut !`
Pour commenter un ensemble de lignes : `/* commentaires */`

6 Les opérateurs les plus usuels

6.1 L'affectation

"v `=` *expr*" où v est une variable (au sens large) et *expr* est une expression, affecte la valeur de *expr* à la variable v **et** retourne la valeur affectée à v comme résultat. Par exemple, "`i = (j = 0)`" affecte 0 à `j` puis à `i` et retourne 0.

6.2 Opérateurs arithmétiques

`*`, `+`, `-`, `/` (division entière et réelle), `%` (modulo).

Remarque 2 *Lors de l'évaluation d'une (sous)expression mélangeant plusieurs types, on utilise le type le plus large. Exemple :*

```
//=========================
int i=3,j=2,m;
double r=3.4;
m = (i/j)*r;
//=========================
```

Le mécanisme d'évaluation se fait de la manière suivante. D'abord l'expression `(i/j)` *est évaluée : comme* `i` *et* `j` *sont entiers,* `/` *désigne la division entière, cela donne donc 1. Puis pour évaluer le produit* `1*r`*, il faut convertir 1 en* `double` *(1.0) et faire le produit sur les doubles, cela donne 3.4, finalement on procède à l'affectation : comme* `m` *est entier, 3.4 est converti en* `int`*. Finalement on a* `m`*=3.*
Une précision : l'évaluation d'une expression ne se fait pas toujours de gauche à droite, par exemple, l'évaluation de `(i/j)*(r/3)` *pourrait commencer par celle de* `(i/j)` *puis* `(r/3)` *ou l'inverse. Ici ca ne change pas le résultat !*
Afin d'éviter des erreurs dues à ce genre de mécanisme de conversion automatique, il est possible de convertir explicitement des données d'un certain type en un autre. Par exemple, dans l'exemple ci-dessus, si on voulait obtenir la division réelle dans `(i/j)` *au lieu de la division entière, il aurait fallu convertir un des deux arguments en* `double`*. Cela peut se faire avec* `double(i)` *L'expression devient alors :* `(double(i)/j)` *et retourne* 1.5*.*

Voir aussi le paragraphe 6.5 pour les priorité entre opérateurs.

6.3 Opérateurs de comparaison

Les opérateurs de comparaison sont :

`<` (inférieur), `<=` (inférieur ou égal), `==` (égal), `>` (supérieur), `>=` (supérieur ou égal) et `!=` (différent). Ces opérateurs sont définis pour les types élémentaires et renvoient 'true' lorsque la condition est vérifiée et 'false' sinon.
Il faut savoir que l'on peut utiliser des valeurs entières au lieu de 'true'et 'false'avec la convention suivante : la valeur 'false' est représentée par l'entier 0 et la valeur 'true' est représentée par n'importe quel entier différent de 0. Chaque instruction conditionnelle (`if`, `while` etc.) utilise cette convention.

Attention à ne pas confondre `=` et `==`

6.4 Opérateurs booléens

`&&` représente l'opérateur "ET" (conjonction), `||` représente le "OU" (disjonction), et `!` représente

le "NON" (négation). Par exemple, l'expression `((x<12) && ((y>0) || !(z>4)))` représente la condition $((x < 12) \wedge ((y > 0) \vee \neg(z > 4)))$.

Lors de l'évaluation d'une expression `e1 && e2` (resp. `e1 || e2`), la sous-expression `e2` n'est évaluée que si `e1` a été évaluée à 'true' (resp. 'false').

6.5 Priorités des opérateurs

La liste ci-dessous classe les opérateurs par ordre de priorité en commençant par les plus prioritaires (ceux figurant au même niveau sont de même priorité). Certains opérateurs ne seront définis qu'aux sections 8 et 9.

- `x[y]  x++  x--`
- `++x  --x  !x  -x`
- `x*y  x/y  x%y`
- `x+y  x-y`
- `x >> y  x << y`
- `x < y  x > y  x >= y  x <= y`
- `x == y  x != y`
- `x && y`
- `x || y`
- `x = y  x op= y`
- `x? y : z`

7 Instructions usuelles

Toutes les constructions suivantes sont des instructions :

- *expr* ;
  où *expr* désigne n'importe quelle expression construite avec les opérateurs vus précédemment.
- { liste d'instructions }
  une telle séquence s'appelle une instruction composée.
- `if (`*expr*`)` *instr*
  `if (`*expr*`)` $instr_1$ `else` $instr_2$
  Exemple d'utilisation :

```
//========================
// où v et i sont des variables
// de type int, float etc.

if (v == 3) i =i+4;

//========================
```

(a)

ou

```
//========================
if ((v==3) && (i<5))
  { i=i+4;
    v=v*2;}
  else v=i;
r = r*3;
//========================
```

(b)

Dans l'exemple (b) ci-dessus, lorsque `v` vaut 3 et `i` est inférieur à 5, alors l'instruction

composée {`i=i+4 ;v=v*2 ;`} est exécutée puis on passe à la dernière instruction `r = r*3 ;`. Si la condition n'est pas vérifiée, on fait `v=i ;` puis `r=r*3 ;`.

- Définitions de variables, de constantes, de classes et de fonctions.
- La figure 1 présente les trois formes de boucles en C++.

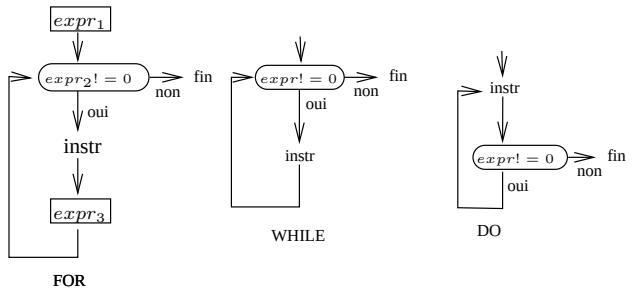

FIG. 1 – Les différentes boucles en C/C++

– `for (`$expr_1$ `;`$expr_2$ `;`$expr_3$`)` *instr*
Par exemple, pour exécuter une instruction *instr* avec les valeurs 0,1,2,... 200 pour la variable `i`, on peut utiliser la structure : `for(i=0 ;i<201 ;i=i+1)` *instr*
Contrairement à certains langages de programmation, la boucle `for` du C/C++ est très générale et peut être utilisée dans de très nombreux cas. De plus, il est possible de composer des expressions avec l'opérateur "`,`" de la manière suivante : `i=i+1,j=j+2` correspond à une instruction qui incrémente `i` de 1, `j` de 2 et qui renvoie la nouvelle valeur de `j` (dernière expression de la liste). Cette possibilité élargit encore le domaine d'application de `for`.
– `while (`*expr*`)` *instr*
– `do` *instr* `while (`*expr*`) ;`
A noter qu'une différence importante entre `while` et `do... while`, c'est que dans le second cas, l'instruction *instr* est toujours exécutée au moins une fois.
On peut aussi remarquer que "`while (`*expr*`)` *instr*" est équivalent à "`for( ;`*expr* `; )` *instr*".

- `break` provoque l'arrêt de la première instruction `do`, `for`, `switch` ou `while` englobant.
- `continue` provoque (dans une instruction `do`, `for`, ou `while`) l'arrêt de l'itération courante et le passage au début de l'itération suivante.
- `switch (`*exp*`) {`
`case` $cste_1$ `:` liste d'instructions$_1$
`...`
`case` $cste_n$ `:` liste d'instructions$_n$
`default :` liste d'instructions
}
Cette instruction permet, selon la valeur de *expr*, d'exécuter la liste d'instructions 1, 2, ... n ou à défaut la dernière liste. Il est important de placer un "`break ;`" à la fin de chaque liste : si il n'y a pas "`break ;`" à la fin de la liste i et que $expr = cste_i$ alors les instructions de la liste i seront exécutées, puis celles de la liste $i + 1$ etc.
[*] Lorsque l'on souhaite définir des variables dans les listes d'instructions correspondant

aux différents cas, il est préférable de le faire au sein d'une expression composée ({ ... }) pour en limiter la portée (voir Section 11), soit en les définissant avant le `switch`.

8 Autres opérateurs

- `(`*exp*`) ?` exp_1 `:` exp_2 vaut exp_1 si *exp* est évaluée à vrai (entier différent de 0) et exp_2 sinon.
- Pré et Post incrément : `++`. L'expression `++`*var* (resp. *var*`++`) incrémente la variable *var* et retourne la nouvelle (resp. l'ancienne) valeur. C'est-à-dire : `++i` équivaut à `i=i+1` alors que `i++` équivaut à `(i=i+1)-1`. Lorsqu'elles sont utilisées en dehors d'une expression, `i++` et `++i` sont donc équivalentes.
- Pré et Post décrément : `--`. L'expression `--`*var* (resp. *var*`--`) décrémente la variable *var* et retourne la nouvelle (resp. l'ancienne) valeur.
- Étant donnée une variable au sens large *var*, une expression *expr* et un opérateur *op* $\in$ {`+,-,*,/,%`}, l'expression "*var op*`=` *expr*" désigne l'expression "*var* `=` *var op expr*". Par exemple :

```
//=========================
int i=2,j=34;
i += 2;
j *= 10;
j *= (i++);
// ici: i==5 et j==1360
//=========================
```

équivaut à

```
//=========================
int i=2,j=34;
i = i+2;
j = j*10;
j = j*i;
i = i+1;
// ici: i==5 et j==1360
//=========================
```

9 Entrées-sorties

Pour utiliser les deux instructions suivantes, il faut ajouter en tête du fichier :

```
# include <iostream>
```

9.1 Afficher à l'écran

Syntaxe : `cout <<` $expr_1$ `<<` ... `<<` $expr_n$ `;`
Cette instruction affiche $expr_1$ puis $expr_2 \ldots$ Afficher un saut de ligne se fait au moyen de `cout << endl`. Par exemple :

```
//=========================
int i=45;
cout << "la valeur de i est "<< i << endl;
//=========================
```

affichera "la valeur de i est 45" suivi d'un retour à la ligne.

Quelques explications (partielles) sur cette instruction :
- `cout` désigne le "flot de sortie" standard,

- `<<` est un opérateur binaire : le premier opérande est `cout` (de type "flot de sortie") et le second opérande est l'expression à afficher, le résultat est de type "flot de sortie" (sa valeur est celle du flot reçu en argument augmenté de l'expression).
- `<<` est associatif de gauche à droite : l'expression ci-dessus est équivalente à `((cout <<` $expr_1$`) << ...) <<` $expr_n$ `;`. La première valeur affichée sera donc bien $expr_1$.
- `<<` est surchargé (ou sur-défini) : on utilise le même opérateur pour afficher des caractères, des entiers, des réels ou des chaînes de caractères.

9.2 Lire au clavier

Syntaxe : `cin >>` var_1 `>>` ... `>>` var_n `;`
Cette instruction lit (au clavier) des valeurs et les affecte [3] à var_1 puis var_2 ...

`cin` est le flot d'entrée standard, et `>>` est un opérateur similaire à `<<`.

Les caractères tapés au clavier sont (après avoir appuyé sur la touche enter) enregistrés dans un tampon dans lequel les `cin` viennent puiser des valeurs. Les espaces, les tabulations et les fins de lignes sont des séparateurs. La lecture d'un chaîne de caractères se fait au moyen de tableaux de caractères. Par exemple, le bout de programme suivant :

```
//========================
string t;                       |
int v;                          |
cin >> v;                       |
cout << "* " << v << endl;      |
for (int i=0;i<4;i++) {         |
  cin >> t;                     |
  cout << "* "<< t << endl;     |
 }                              |
//========================
```

attendra une lecture au clavier, puis si on tape la chaîne "1234 bonjour tout le monde" suivie de enter, le programme affichera :

```
* 1234
* bonjour
* tout
* le
* monde
```

NB : le mieux, comme toujours, est d'essayer `:-)`

10 Fonctions

Définition d'une fonction : *type nom*(liste des paramètres){corps}

- *type* est le type du résultat renvoyé par la fonction (on utilise `void` lorsqu'il s'agit d'une procédure, i.e. lorsqu'il n'y a pas de valeur de retour).
- La liste des paramètres (appelés *paramètres formels*) est de la forme : $type_1\ p_1, \ldots, type_n\ p_n$, signifiant que le premier paramètre s'appelle p_1 et est de type $type_1$ etc.
- Le corps de la fonction décrit les instructions à effectuer lors de l'appel de cette fonction. Le corps utilise ses propres variables locales (à définir), les éventuelles variables globales et les paramètres formels (considérés comme des variables locales).
- Lorsqu'une fonction renvoie un résultat, il doit y avoir (au moins) une instruction "`return` *expr* `;`" où *expr* est une expression du type de la fonction. Cette instruction met fin à

[3] var_i représente des variables au sens large.

l'exécution de la fonction et retourne *expr* comme résultat. Dans une procédure, il est possible (non obligatoire) d'utiliser l'instruction `return ;` (ici sans argument) pour mettre fin à l'exécution de la procédure.

Exemple :

```
//=========================
int  max(int a,int b)
{ int res=b;
  if (a>b)  res = a;
  return res;
}
//=========================
```

Appel d'une fonction : *nom*(liste des arguments)
La liste des arguments (aussi appelés *paramètres réels*) est de la forme : $expr_1, expr_2, \ldots expr_n$ où chaque $expr_i$ est compatible avec le type $type_i$ du paramètre formel p_i. Exemple :

```
//=========================
int k=34, t=5, m;
m = max(k,2*t+5);
//=========================
```

Il est possible de définir des fonctions récursives :

```
//=========================
int fact(int n)
{ if (n <= 1)
   return 1;
  else return n*fact(n-1); }
//=========================
```

Un appel de fonction doit **toujours** être précédé par la définition ou une déclaration de la fonction. Une déclaration consiste à donner : son type, son nom et le type des paramètres (le nom des paramètres n'est pas nécessaire) suivi d'un point virgule :
Déclaration d'une fonction : *type nom*(liste des types des paramètres);
Par exemple : "`int max(int ,int ) ;`". Une déclaration permet au compilateur de vérifier que les différents appels sont corrects du point de vue du type des arguments et du type du résultat. A noter qu'une fonction doit être définie une et une seule fois mais peut être déclarée autant de fois qu'on le désire.

De plus, il est possible d'utiliser le même nom pour plusieurs fonctions si la liste des paramètres est différente (nombre ou type différent). Concernant le passage des paramètres, deux possibilités existent : le *passage par valeur* (standard) et le *passage par référence*.

10.1 Passage des paramètres par valeur.

C'est le cas standard : les paramètres sont initialisés par une copie des valeurs des paramètres réels. Modifier la valeur des paramètres formels dans le corps de la fonction ne change pas la

valeur des paramètres réels. Par exemple, l'exécution du bout de programme à gauche avec la définition de `proc` à droite :

```
//=========================
int k=10;
proc(k);
cout<<"la valeur de k est "<<k;
//=========================
```

```
//=========================
void proc(int a)
{
a++;
cout<<"la valeur de a est "<<a;
cout << endl;
}
//=========================
```

entraînera l'affichage de "`la valeur de a est 11`" suivi d'un retour à la ligne puis de "`la valeur de k est 10`".

10.2 Passage des paramètres par référence.

Il est parfois nécessaire d'autoriser une fonction à modifier la valeur d'un paramètre réel. Pour cela, il est possible d'utiliser des *références*.

Cette notion de référence n'est pas limitée au passage de paramètres. De manière générale, une référence sur une variable est un synonyme de cette variable, c'est-à-dire une autre manière de désigner le même emplacement de la mémoire. On utilise le symbole `&` pour la déclaration d'une référence. Dans l'exemple ci-dessous :

```
int i = 4;
int & j = i;
```

nous avons déclaré `j` comme étant un synonyme de `i` : modifier `j`, c'est modifier `i`. La principale utilisation des références concernent le passage des paramètres. Dans la liste des paramètres formels d'une définition de fonction, "*type* `&` p_i" déclare le paramètre p_i comme étant une référence sur le i^{eme} paramètre réel v_i : p_i n'est donc plus une variable créée pour l'exécution de la fonction et initialisée avec v_i, mais p_i est un synonyme de v_i, modifier p_i revient à modifier v_i. Par exemple, l'exécution du bout de programme ci-dessous avec la définition de la procédure `proc` à droite :

```
//=========================
int k=10;
proc(k);
cout<<"la valeur de k est "<<k;
//=========================
```

```
//=========================
void proc(int & a)
{
a++;
cout<<"la valeur de a est "<<a<<endl;
}
//=========================
```

entraînera l'affichage de "la valeur de a est 11" suivi de "la valeur de k est 11".

Autre exemple :

```
//=========================
void permut(int & a, int & b)
{
int aux = a;
a = b;
```

```
b = aux;
}
//========================
```

La fonction `permut` permet d'échanger la valeur des deux variables passées en argument. Il faut noter que le passage par référence demande de passer des variables (au sens large) en paramètres réels : on en peut pas appeler `permut(i,3)`. On peut noter qu'il existe une fonction `swap` qui permute la valeur de deux arguments (de type quelconque) dans la bibliothèque `algorithm` (à utiliser avec `#include <algorithm>`).

Remarque 3 [*] *En général, le passage par référence est plus efficace du point de vue de la gestion de la mémoire dans la mesure où il évite la recopie des arguments* [4] *(car seule l'adresse mémoire de l'argument est communiquée à la fonction) mais il est dangereux car des fonctions peuvent modifier les données. Il est donc quelquefois intéressant de passer des paramètres en tant que référence sur une constante (notation* `const` *type* `&` *p) : cela permet d'éviter le coût de la recopie des arguments et reste sûr car le compilateur vérifiera que le paramètre p n'est pas modifié dans le corps de la fonction. Cette technique est très largement utilisée et peut être vue comme une troisième manière de passer des arguments.*
Pour finir, on peut noter qu'il est aussi possible pour une fonction de renvoyer comme résultat une référence.

10.3 Valeurs par défaut [*]

On peut attribuer des valeurs par défaut aux paramètres d'une fonction. Par exemple :

```
//========================
int max(int a,int b =0)
{
  return (a>b) ? a : b;
}
//========================
```

On peut alors appeler `max(i)` au lieu de `max(i,0)`. Il faut noter la restriction suivante : si un paramètre a une valeur par défaut, tous les paramètres suivants en ont une aussi.

10.4 Fonctions en ligne [*]

Pour de petites fonctions, il peut être plus efficace (rapide) de se dispenser d'un appel de fonction et d'expanser le corps de la fonction en lieu et place de chaque appel. C'est possible avec le mot clé `inline`. Par exemple, on pourra définir `max` de la sorte :

```
//========================
inline int max(int a,int b =0)
{
  return (a>b) ? a : b;
```

[4] Cet argument n'est justifié que pour des structures de données de taille importante, par exemple les `vector`.

```
}
//=========================
```

Attention : ce choix n'est intéressant que pour de petites fonctions... A utiliser avec parcimonie !

10.5 Structure générale d'un programme C++

Un programme C++ est réparti dans un ou plusieurs fichiers, chacun contenant des définitions/déclarations de fonctions, des définitions de types (abordés plus loin) et des définitions de variables globales. Il existe une (seule) fonction `main`, elle correspond à la fonction qui sera exécutée après la compilation. Lorsqu'il n'y a pas de paramètre à la fonction `main`, son profil est `int main()`. Si l'on veut passer des arguments avec la ligne de commande, on définit `main` avec le profil `int main( int argc, char ** argv )`, `argc` contient le nombre de paramètres donnés, et `argv` est un tableau de chaînes de caractères (voir la section 17) où `argv[0]` désigne le nom de la commande utilisée pour lancer le programme, `argv[1]` désigne le premier paramètre, `argv[2]` le second etc.

Le programme ci-dessous est un programme complet avec deux fonctions :

```
//=========================
#include <iostream>

void test(int j)
{ cout << j << endl; }

int main()
{
int i =20;
cout << "bonjour" << endl;
test(i);
}
//=========================
```

Remarque 4 $\boxed{*}$ *Un programme important utilise de nombreuses bibliothèques, fonctions etc. Pour éviter les problèmes de conflit de noms, on utilise des* espaces de noms *(*namespace*) : on associe un nom à un ensemble de variables, types, fonctions. Pour désigner ces différents objets, il faut alors utiliser leur nom précédé par leur nom d'espace suivi de* `::`*. Par exemple, la fonction* `int fct(){...}` *définie dans le domaine* `A` *aura pour nom complet* `A::fct()`*. Pour les entrées-sorties avec* `cout` *et* `cin`*, il faut en fait utiliser* `std::cout`*,* `std::cin` *et* `std::endl` *(i.e. les objets* `cout`*,* `cin` *et* `endl` *de la bibliothèque standard) pour être parfaitement correct (le compilateur tolère encore la notation simple mais...). C'est aussi le cas pour* `std::string` *et* `std:vector`*. Comme il est pénible de mentionner systématiquement le nom d'un espace, on peut utiliser la commande* `using` *pour préciser que les objets non-explicitement définis dans le code viennent de certains espaces de noms. Par exemple, tous nos programmes devraient (et doivent à partir de maintenant) comporter l'instruction suivante "*`using namespace std;`*".*

11 Portée, visibilité, durée de vie

La *portée* d'un identificateur (correspondant à un nom de variable, de fonction,...) correspond aux parties du programme où cet identificateur peut être utilisé sans provoquer d'erreur à la compilation.

La portée d'une variable globale ou d'une fonction globale est égale au programme : elles peuvent être utilisée n'importe où. Un identificateur correspondant à une variable locale peut être utilisé à partir de sa définition jusqu'à la fin de la première instruction composée ({...}) qui contient sa définition.

Deux variables peuvent avoir le même nom mais dans ce cas, elle doivent avoir deux portés différentes. Par exemple, on peut évidemment utiliser une variable locale `i` dans deux fonctions différentes : dans ce cas, les portées des deux variables sont disjointes. Mais on peut aussi avoir la situation suivante :

```
//========================
{
 int i=3;
  {
   int i=5;
   cout << i; // affiche 5
  }
cout << i; // affiche 3
}
//========================
```

Ici les deux variables `i` ont bien des portées différentes : la portée de la seconde est incluse dans celle de la première. Lors du premier `cout`, seule la seconde variable (celle valant 5) est *visible*, la première existe toujours (elle occupe un espace mémoire) mais elle n'est pas accessible. Après le }, la seconde variable est détruite et il ne reste plus que la première variable valant 3. Une variable peut donc être "à portée" mais non visible.

La *durée de vie* d'une variable correspond à la période depuis sa création (allocation de mémoire) à sa destruction (libération de la zone mémoire correspondante). Cette notion coïncide avec la portée pour toutes les variables que nous avons évoquées jusqu'ici, mais ce n'est pas le cas pour les variables `static` et pour les celles correspondant à des structures de données dynamiques (i.e. dont la création n'a pas été effectuée avec l'opérateur `new`), ces deux cas seront présentés ultérieurement.

12 Divers

12.1 La librairie `cassert` *

La bibliothèque `cassert` permet d'insérer dans les programmes des commandes "`assert(`*expr*`) ;`" où *expr* est une expression booléenne : cette instruction, lorsqu'elle est rencontrée au cours de l'exécution, provoque l'arrêt du programme si *expr* est interprétée à FAUX. Cela permet de garantir simplement des conditions d'entrée et de "débugger" plus aisément les programmes.

12.2 GDB *

Pour la mise au point des programme, il est possible (et recommandé si le programme est gros) d'utiliser GDB (The GNU Debugger). Pour l'utiliser vous pouvez :

– Faire `man gdb` dans une fenêtre shell.
– Voir le site `http ://www.linux-france.org/article/devl/gdb.html`
– Chercher d'autres infos sur GDB avec Google.

B. Classes C++

Les types élémentaires ne sont pas suffisants : lorsqu'on implémente un algorithme, on souhaite garder le plus possible la structure initiale de l'algorithme et donc utiliser les diverses structures de données classiques (pile, file, arbre, graphe etc.) et de nouvelles structures ciblées pour son application. Il faut donc pouvoir définir de nouveaux *types de données* (incluant des fonctions de manipulation, par exemple "Empiler", "Dépiler" etc. pour les piles).

13 Regroupement de données - `struct`

La première étape consiste à définir des types correspondant à des regroupements de données, des n-uplets. Par exemple, une fraction peut être vue comme une paire d'entiers : un numérateur et un dénominateur. Nous pouvons définir un type lui correspondant de la manière suivante :

```
struct fraction {
 int num;
 int den;
};
```

Il devient alors possible de définir des variables de type `fraction` par l'instruction habituelle : `fraction f, g ;`
La variable `f` est une structure de deux entiers, on parle de *champs* : un champ `num` et un champ `den`. On peut accéder aux champs avec l'opérateur "`.`" : `f.num` ou `f.den`. Ces deux champs peuvent s'utiliser comme n'importe quelle variable entière :

```
//========================
f.num = 1;
f.den = 3;
cout << f.num << "/" << f.den;
//========================
```

On peut évidemment définir d'autres champs :

```
struct envrac {
 int val;
 double d1, d2;
 bool b;
};
```

Une variable `e` de type `envrac` contient un champ entier (`e.val`), deux champs flottants (`e.d1` et `e.d2`) et un champ booléen (`e.b`).

Une expression "`g = f`" où `f` et `g` sont d'un type `struct`, correspond à l'affectation "champ à champ" : la valeur de chaque champ de `f` est recopiée dans le champ correspondant de `g`. Ce mécanisme est aussi appliqué lorsque l'on utilise des objets de type `struct` en paramètres de fonction ou lorsque on utilise ce type comme résultat d'une fonction.

14 Classes C++

Les regroupements de données obtenus par `struct` permettent de définir de nouveaux types mais restent insatisfaisants. D'une part, ils n'intègrent pas directement les fonctions de manipulation du type de données que l'on souhaite implémenter. D'autre part, il est toujours possible d'accéder directement aux différents champs des objets des `struct`, ce qui rend les programmes peu sûrs et peu adaptables : le choix du codage (le nom et le type des champs) ne devrait intervenir que dans la définition des fonctions de manipulation (pour le type fraction : addition, produit etc.) et les autres fonctions du programme devraient manipuler les objets de notre type uniquement au moyen de ces fonctions de manipulation. Ainsi, un changement de codage n'entrainerait que des changements locaux dans le code source. Les classes C++ permettent cela.

Une classe C++ est un regroupement de données (champs) et de fonctions (appelées *fonctions membres* ou *méthodes*) sur ces données. Il est aussi possible de décider pour chacun de ces éléments (données ou fonctions) si il est accessible, ou non, aux autres fonctions extérieures à la classe.

Prenons l'exemple d'une pile d'entiers. Un objet de ce type doit être muni des opérations suivantes : "empiler un entier", "récupérer la valeur du dernier entier empilé" (la tête de la pile), "dépiler" et "tester si la pile est vide". Nous considérons que nos piles sont de taille bornée (cette borne étant fixée à la création de la pile). Nous pouvons utiliser un tableau d'entiers pour stocker les éléments. La classe C++ correspondante pourrait être [5] :

```
//========================
class  PileE {
 private :
   vector<int> Tab;
   int NbElem;       //  nb d'éléments dans la pile
 public :
   PileE(int Max);
   void Empiler (int val);
   int Tete();
   void Depiler();
   int EstVide();
};
//========================
```

Un objet `P` de type `PileE` serait donc composé de deux champs de données : un tableau

[5] La classe `vector` dispose déjà de toutes les opérations voulues (empiler, dépiler, etc.), cet exemple est donc très artificiel mais éclairant... :-)

`P.Tab` pour stocker le contenu de la pile et un entier `P.NbElem` correspondant au nombre d'éléments présents dans `P` à un moment donné. De plus, l'objet `P` est muni de 5 fonctions membres (ou *méthodes*) qui sont déclarées dans la définition de la classe. Laissons de coté, pour le moment, la première fonction (appelée constructeur) `PileE(int Max))`. Le mécanisme d'appel de ces fonctions se fait de la manière suivante "`P.Empiler(`*expr*`)`" (où *expr* renvoie un entier), "`P.Tete()`", "`P.Depiler()`" ou encore "`P.EstVide()`". Les fonctions membres sont donc appelées **sur** un objet de la classe (on parle aussi d'appliquer une méthode à un objet), c'est donc un argument supplémentaire qui n'apparaît pas comme les autres dans le profil de la fonction [6]. Le nom complet de ces fonctions est :

```
void PileE::Empiler(int val);
int  PileE::Tete();
void PileE::Depiler();
int  PileE::EstVide();
```

Leur définition utilise les champs et les méthodes de l'objet sur lequel les fonctions seront appelées. Ainsi on peut définir [7] :

```
//========================
void PileE::Empiler(int val) {
if (NbElem >= Tab.size())
    { cout << "Erreur !" << endl; exit(1);}
Tab[NbElem]=val;
NbElem++;}

void PileE::Depiler() {
if (NbElem==0)
    {cout << "Erreur !" << endl; exit(1);}
NbElem--;}

int PileE::Tete() {
if (NbElem==0)
    {cout << "Erreur !" << endl; exit(1);}
return Tab[NbElem-1];}

int PileE::EstVide()
{return (NbElem == 0);}
//========================
```

L'idée de ce codage est de stocker le contenu de la pile dans `Tab[0], ..., Tab[NbElem-1]`, le dernier élément empilé se trouvant en `Tab[NbElem-1]`. La fonction `PileE::Empiler` commence par tester si il reste de la place dans la pile, puis affecte le nouvel élément dans la première case libre du tableau et incrémente `NbElem` (les deux champs `Tab` et `NbElem` dans ces définitions correspondent aux champs de l'objet sur lequel ces fonctions seront utilisées dans le programme). Avec ces définitions, nous pouvons maintenant utiliser des piles d'entiers :

[6] En fait, la longueur d'une chaîne de caractères `s.size()` correspond à l'appel de la fonction membre `size()` sur l'objet `s` de la classe `string`.

[7] `exit(1)` est une instruction arrêtant brutalement l'exécution du programme, elle est utilisée ici en cas d'erreur. Nous verrons d'autres solutions plus "propres".

```
//=========================
int main() {
int i,v;
PileE P(100);  // appel du constructeur avec 100 comme argument

for (i=0;i<20;i++) {
  cin >> v;
  P.Empiler(v);
  }
while (! P.EstVide()) {
  cout << P.Tete();
  P.Depiler();
  }
}
//=========================
```

Ce petit programme se contente de lire 20 nombres et les empile dans `P`, puis les dépile en les affichant au fur et à mesure... Il aurait été impossible d'utiliser directement les champs `Tab` et `NbElem` de `P` dans la procédure `main` ci-dessus car ces deux champs sont déclarés `private` et ne sont donc accessibles qu'aux fonctions membres de la classe `PileE`.

Il reste un problème concernant l'initialisation de la variable `P`. En effet, pour que le programme ci-dessus soit correct il faut que le tableau `P.Tab` soit construit (avec une certaine taille) et que le champ `P.NbElem` soit initialisé avec la valeur 0 lors de la création de l'objet `P`, ce type d'initialisation est le rôle du ou des *constructeurs* de la classe. Dans notre exemple, il existe un seul constructeur : `PileE(int Max)` (dont le nom complet est `PileE::PileE(int Max)`). Le nom d'un constructeur est celui de la classe et il n'a pas de type car un constructeur retourne toujours un objet de sa classe. Un constructeur est automatiquement appelé lorsqu'un objet de la classe est créé. l'instruction "`PileE P(100) ;`" entraîne son exécution avec 100 comme argument. Il est aussi possible d'invoquer un constructeur comme une autre fonction, par exemple "`P = PileE(10) ;`" est correct. L'objectif des constructeurs est d'initialiser les champs correctement. Ici nous devons créer une pile de taille `Max` et mettre `NbElem` à 0 :

```
//=========================
PileE::PileE(int Max)
{   Tab = vector<int>(Max);
    NbElem = 0; }
//=========================
```

La notation utilisée pour la création du tableau vient du fait que nous devons créer un objet de la classe `vector` et pour cela on appelle un de ces constructeurs (`vector<int>( , )`) de manière explicite.

14.1 Champs et méthodes

Le schéma général de la définition d'une classe est celui-ci :

```
//========================
class  MaClasse {
 private :
    // definition de champs et methodes prives:

    type Nom-de-champ;
    type Nom-de-fct (liste parametres);
    type Nom-de-fct (liste parametres) { corps }

 public :
    // definition des constructeurs:
    MaClasse(liste parametres1);
    MaClasse(liste parametres2) {corps}

    // definition de champs et methodes publics:
    type Nom-de-champ;
    type Nom-de-fct (liste parametres);
    type Nom-de-fct (liste parametres) { corps }
};
//========================
```

Ce schéma appelle plusieurs remarques :

- Un champ ou une méthode peut être `private` ou `public`. Une méthode privée ne peut être appelée que depuis une autre méthode de la classe [8].
- La définition (le corps) des méthodes peut se faire directement dans la définition de la classe. On réserve cela aux fonctions membres très simples. En général, Il est préférable de déclarer la méthode dans la classe puis de la définir à l'extérieur avec une définition de la forme :
  ```
  type MaClasse::Nom-de-fct (liste parametres)
  { corps }
  ```

14.2 Constructeurs et destructeurs

- Les constructeurs sont des méthodes particulières appelées automatiquement à chaque définition d'un objet de la classe, c'est-à-dire à chaque fois que de la mémoire est allouée pour stocker un élément de la classe. Si il existe plusieurs constructeurs, ils doivent avoir une liste de paramètres différents [9].
- Comment passer des arguments à un constructeur ? Il suffit de faire suivre le nom de la variable (au sens large) par la liste des arguments : "`MaClasse A(i,f) ;`" définit une variable `A` et appelle le constructeur à deux arguments compatible avec les types de `i` et `f`. Si un tel constructeur n'existe pas, il y aura une erreur à la compilation. Lorsqu'aucun argument n'est donné, le constructeur sans argument est appelé.
- [*] Lorsqu'aucun constructeur n'est défini explicitement, le compilateur crée un constructeur sans argument `MaClasse::MaClasse()` qui se contente d'appeler les constructeurs

[8] Il est possible de contourner cette limitation avec les fonctions et les classes "amies" (`friend`), voir section 14.3.

[9] comme toute fonction C++ : on peut avoir plusieurs fonctions de même nom à condition d'avoir des paramètres différents.

(sans argument) de chaque champ de l'objet. Mais il suffit d'avoir défini un seul constructeur pour que le constructeur sans argument ne soit pas créé automatiquement, il faut alors le définir explicitement pour pouvoir définir des variables sans donner de paramètres. Par exemple, dans notre exemple de pile, il n'est pas possible d'écrire "`PileE P;`", car il n'existe pas de constructeur "`PileE::PileE()`". Il serait facile de résoudre ce point en adaptant "`PileE::PileE(int Max)`" en donnant une valeur par défaut à `Max`.

- [**] Il existe un constructeur particulier appelé "constructeur copieur" de la forme [10] `MaClasse (const MaClasse &)` qui est utilisé pour le passage des paramètres et les initialisations :
  - Lorsqu'une fonction a un paramètre `A` du type `MaClasse`, un appel de cette fonction avec une variable `V` de `MaClasse` conduit à l'*initialisation* de `A` avec `V` via le constructeur copieur.
  - Lorsqu'on définit une variable `V` et qu'on l'initialise avec une valeur `E`, par l'instruction "`MaClasse V=E ;`", le processus sous-jacent est différent de celui mis en oeuvre lorsque l'on définit `V` ("`MaClasse V ;`") et qu'ensuite on lui affecte la valeur `E` ("`V=E ;`") : le premier cas consiste en une *initialisation* et à l'appel du constructeur copieur, le second cas consiste en une *affectation* et un appel de l'opérateur `=` (voir section 15)... Ça peut paraître curieux mais c'est comme ça ! Ce genre de subtilité peut être oublié en première lecture car...

  Le constructeur copieur créé par défaut se contente de faire une initialisation champ à champ avec les valeurs des champs de la variable donnée en argument. Ça tombe bien, c'est en général ce qu'il nous faut...

- [*] De même qu'il existe des constructeurs mis en oeuvre pour la création de nouveaux objets, il est possible de définir un (et un seul) destructeur qui est appelé automatiquement lorsqu'une variable locale est détruite (par exemple à la fin d'une fonction). Les destructeurs sont surtout utiles lorsqu'on manipule des structures de données dynamiques (section 19).

Le plus simple pour comprendre les différents mécanismes d'appel, c'est de définir une classe jouet et de (re)définir différents constructeurs en y plaçant des "`cout <<`" pour signaler leur mis en oeuvre, et de tester le tout avec un petit programme. Un exemple se trouve sur la page web.

14.3 Fonctions amies [**]

Déclarer une fonction F comme étant "amie" (`friend`) d'une classe C1, c'est autoriser, dans la définition de F, l'accès aux champs privés de C1. On peut aussi déclarer une classe C2 comme étant "amie" de la classe C1 ; dans ce cas, toutes les fonctions membres de C2 peuvent accéder aux champs privés de C1. Ces déclarations se font dans la définition de F :

```
class C {
   ...
  friend type-de-F F(param-de-F);
  friend class C2;
   ...
```

[10] L'utilisation de la forme `const &` est utilisée pour des raisons d'optimisation (voir remarque 3).

```
};
```

14.4 Conversions entre types **

La section 6.2 présente des conversions implicites ou explicites entre différents types élémentaires. Il peut être utile de faire de même pour les types que l'on définit.

Certains constructeurs définissent des fonctions de conversion : un constructeur de `T1` qui prend un unique paramètre de type `T2` définit implicitement une conversion de `T2` dans `T1`. Dans ce cas, une fonction prenant un paramètre de type `T1` pourra être appelée avec un argument de type `T2` sans provoquer d'erreur de compilation : l'argument sera converti en `T1` puis la fonction sera exécutée. Néanmoins ce mécanisme (de conversion implicite) ne se compose pas : le compilateur ne cherche que les conversions en une étape (sauf pour les types élémentaires).

Par exemple, une classe fraction qui aurait un constructeur prenant un entier en paramètre (par ex. dans le but de renvoyer une fraction avec le dénominateur 1) définit une conversion des `int` vers les `fraction` et une fonction prenant deux paramètres de type `fraction` en argument, peut être appelée avec une fraction et un entier, ce dernier sera correctement converti. Un exemple de programme avec conversions est donné sur la page web.

Le mécanisme de conversion mentionné ci-dessus nécessite d'accéder à la définition du type cible `T1`. Ce n'est pas toujours possible, par exemple lorsqu'on utilise un type élémentaire. Dans ce cas, on définit une fonction membre `operator T1()` dans la classe `T2`. Comme un constructeur, la déclaration de cette fonction ne fait pas apparaître de type, là encore le type est implicite, il s'agit de `T1`. Par exemple :

```
class fraction {
private :
   int num, den;
public :
   ...
   fraction(int v)
     {num=v;den=1;}
   double()
     {return double(num)/den;}
};
```

Ici le type fraction est muni de deux fonctions de conversions implicites ; une de `int` vers `fraction`, la seconde de `fraction` vers `double`. Évidemment, plus on définit de conversions, plus le risque d'avoir des expressions ambigues (où plusieurs choix sont possibles) augmente... Il est possible de signifier au compilateur qu'un constructeur à un argument ne doit pas être utilisé dans les conversions, pour cela il faut ajouter le mot clé `explicit` devant la déclaration du constructeur.

15 Surcharge d'opérateurs *

Lorsqu'on définit un type fraction, on souhaite évidemment disposer des opérations arithmétiques de base sur ces objets, on souhaite même pouvoir écrire dans un programme des expressions `g = f*h+f` où `f`,`g` et `h` sont de type `fraction`. On souhaite donc donner une nouvelle définition (surcharger) aux opérateurs `+` et `*`. C'est possible !

Seuls les opérateurs déjà définis peuvent être surchargés. De plus, les arités et les priorités ne changent pas. La liste des opérateurs est longue, voir [1, 3]. La surcharge d'opérateurs peut se faire selon deux points de vue : comme fonction globale ou comme fonction membre.

15.1 Approche "fonction globale"

Etant donné *op* un opérateur unaire (par exemple `+`, `-`, `!`, `++` etc.), l'expression "*op* `A`" peut être vue comme la fonction globale `operator` *op*`(T a)` appliquée à l'objet `A`.
Etant donné, *op* un opérateur binaire (par exemple `+`, `-`, `*`, `&&` etc.), l'expression "`A` *op* `B`" peut être vue comme la fonction globale `operator` *op*`(T a, T b)` appliquée à l'objet `A` et à l'objet `B`. Par exemple, on pourrait définir :

```
fraction operator -(fraction f)           // def. operateur unaire -
   { return fraction(- f.Numerateur(),f.Denominateur());}

fraction operator *(fraction f, fraction g)      // def. operateur binaire *
   { return fraction(f.Numerateur()*g.Numerateur(),
                     f.Denominateur()*g.Denominateur());}
```

Ces fonctions sont globales, elle ne sont donc pas définies (ni déclarées) dans la classe mais en dehors. Elles n'ont donc pas accès aux champs privés de `fraction` (à moins d'avoir été déclarée fonctions amies) et nous devons donc utiliser (et définir) des fonctions de lecture des champs `num` et `den` dans la classe `fraction`.

15.2 Approche "fonction membre" **

Étant donné *op* un opérateur unaire, une expression "*op* `A`" où `A` est de type `T` peut être vue comme l'objet `A` sur lequel est appliqué la fonction membre `operator` *op*`()`. De même, l'expression "`A` *op* `B`" peut être vue comme l'objet `A` sur lequel est appliqué la fonction membre `operator` *op*`(T b)` avec `B` comme argument. Par exemple, on pourrait avoir :

```
class fraction {
private :
  int num,den;
public :
  ...
  fraction operator -();
  fraction operator *(fraction h);
};
```

```
fraction fraction::operator -()              // def. operateur unaire -
  {return fraction(-num,den);}

fraction fraction::operator *(fraction h)    // def. operateur binaire *
  {return fraction(num*h.num,den*h.den);}
```

15.3 Lequel choisir ?

En général, il est recommandé d'utiliser les fonctions membres pour surcharger les opérateurs unaires. Pour les opérateurs binaires, on distingue deux cas : si l'opérateur a un effet de bord sur un des opérandes (par exemple, `+=`), il est préférable d'utiliser des fonctions membre, cela souligne le fait qu'un opérateur est appliqué sur un opérande. Pour les autres cas, les fonctions globales sont souvent préférables pour utiliser des conversions ; par exemple, si `f` est de type `fraction` et qu'il existe des conversion `int -> fraction`, alors `f+3` et `3+f` seront possibles si `+` est une fonction globale, mais pas si on utilise une fonction membre...

Cependant certains opérateurs (par exemple, `=`, `->`, `()`,`[]`) ne peuvent être surchargés que par le biais d'une fonction membre.

15.4 Surcharger `cout` ≪

L'opérateur `<<` ne peut être surchargé que de manière globale, le profil de cet opérateur pour un type `T` est " `ostream & << (ostream &, T )`", c'est à dire une fonction recevant en argument un flot de sortie et une valeur de type `T` et retournant un flot de sortie (augmenté de la nouvelle valeur). Le fait de retourner une valeur de flot permet d'emboiter les `<<` successivement.

Dans le cas des fractions, on pourrait définir :

```
//-----------------------------------
ostream & operator <<(ostream & os,fraction f)
{
os << f.Numerateur() << "/" << f.Denominateur();
return os;
}
//-----------------------------------
```

La surcharge de l'opérateur est `>>` est plus délicate : voir [3].

C. Pointeurs - structures de données dynamiques

Les variables permettent de désigner des cases mémoires au moyen de nom symbolique. Il est aussi possible de désigner directement les emplacements en mémoire par leur *adresse*. Un pointeur permet de stocker ces adresses mémoire.

16 Pointeurs, *, &

Un pointeur est typé : on distingue les pointeurs sur un `int`, sur un `double`, sur un objet de type `fraction` etc. La définition d'un pointeur se fait comme suit :
Définition d'un pointeur : *type* `*` *nom* ;
nom est un pointeur sur *type*, i.e. pouvant recevoir une adresse désignant un objet de type *type*. Il faut noter que le symbole `*` s'applique à *nom* : "`int *p, i;`" définit un pointeur `p` sur `int` et une variable `i` de type `int`. L'adresse 0 est utilisée pour signifier qu'un pointeur ne pointe sur rien. Noter qu'un pointeur sur un type `T` est une variable comme une autre, ayant sa propre adresse qu'il est possible de stocker dans un pointeur de type "pointeur sur pointeur sur `T`" (i.e. "`T * *`") etc. Il y a deux opérations fondamentales :

- Étant donnée une variable `V`, il est possible d'obtenir l'adresse de `V` avec l'opérateur `&` : `&V` représente l'adresse mémoire de `V`.
- Étant donné un pointeur `P`, il est possible de désigner l'objet pointé par `P` (i.e. l'objet situé à l'adresse contenue dans `P`) au moyen de l'opérateur (unaire) `*` : "`*P`" désigne l'objet pointé par `P`. Par exemple :

```
int i, *pi, k;
i=3;
pi = &i;       // pi pointe sur i
k = (*pi)+5;   // identique à :   k=i+5
(*pi) = k+2;   // identique à :  i = k+2
(*pi)++;
// ici, i=11, k=8
```

Connaissant l'adresse d'un objet, il est donc possible d'utiliser sa valeur et de la modifier. Cela offre une autre manière de passer des arguments à des fonctions pouvant les modifier : au lieu du passage par référence, il est possible de passer (par valeur) des adresses et de modifier les données contenues à ces adresses via l'opérateur `*`. La fonction `permut` de la section 10.2 pourrait s'écrire :

```
//========================
void permut(int * pa, int * pb)
{
int aux = (*pa);
(*pa) = (*pb);
(*pb) = aux;
}
```

L'appel de cette fonction `permut` doit prendre deux adresses : pour échanger les valeurs des variables `i` et `j`, on écrirait "`permut(&i,&j) ;`". Cette méthode (passage des adresses) est la seule possible en C (le passage par référence n'existe pas). En fait, les mécanismes internes sont similaires.

17 Pointeurs et tableaux

A la section 4, nous avons vu comment définir des tableaux avec le type `vector`. Il existe une autre manière, plus basique (on les appelera *tableaux simples*), pour manipuler des tableaux :

Syntaxe : *type* *nom*`[`*taille*`] ;`
définit un tableau d'éléments de type *type* et de taille *taille* (les indices vont de 0 à *taille* − 1). La taille doit être une constante. L'élément d'indice i est désigné par *nom*`[`i`]`. Par exemple :

```
int Tab[100];
double M[10];
```

On peut définir des tableaux à plusieurs dimension, par exemple : `int A[10][50] ;`

Il n'est pas possible de récupérer la taille de ces tableaux : il n'existe pas de fonction `size()`. Mais (contrairement aux `vector`) on peut les initialiser lors de leur définition :

```
double t[5] = {1.34, 2.21, 3.0, 4.34, 5.78};
char ch[3] = {'a','l','v'};
int t[4][3] = { {0,1,2}, {3,4,5}, {6,7,8}, {9,10,11}};
```

De plus, lorsqu'on définit un tableau par l'instruction "`MaClasse Tab[N] ;`", le constructeur sans argument est appelé sur les `N` cases du tableau. Ceci n'est pas vrai pour les types élémentaires : il faut explicitement initialiser les éléments des tableaux de `int`, `double` ...

Tableaux = pointeurs. Dans la définition "`int Tab[10]`", le terme `Tab` désigne en fait l'adresse de la première case du tableau, c'est-à-dire "`&Tab[0]`" et le type de `Tab` est "`const int *`", `Tab` est donc un pointeur (constant) sur un entier.

Qu'est-ce que `Tab[i]` ? `Tab[i]` représente la `i`-ème case de `Tab`, c'est à dire `(*(Tab+i))` : le contenu (`*`) de la `i`-ème case après l'adresse `Tab`. Notez que la taille d'une case dépend du type [11] (par ex. une fraction prend plus de place en mémoire qu'un entier) ; néanmoins le compilateur, connaissant le type de `Tab`, sait ce que représente le décalage de `i` cases. Considérons les quelques lignes ci-dessus :

```
//============================
int T[10];
int * p;
p = &T[4];
p[0] = 100;  //  équivaut à T[4] = 100
p[1] = 200;  //  équivaut à T[5] = 100
//============================
```

Les équivalences mentionnées sont dues au fait que "`p[i] = *(p+i)`" et donc "`p[i] = *(t+4+i)`".

Tableaux et fonctions. Passer un tableau (simple) en paramètre d'une fonction revient à passer l'adresse de la première case, et donc les tableaux simples sont toujours passés par référence : si `T` est un paramètre formel correspondant à un tableau, modifier `T[i]` modifiera le i^{eme} élément du paramètre réel (un tableau) correspondant.
Pour déclarer un paramètre `T` de type tableau (par ex. d'entiers), on utilise la notation : `int T[]`. La taille (si nécessaire) doit être passée en tant que telle avec un autre paramètre. Par exemple : `void Tri(int T[],int taille)`

[11] La taille d'un type peut s'obtenir avec la fonction `sizeof` : "`cout≪ sizeof(int) ;`" affichera le nombre d'octets nécessaires au stockage d'un entier.

D'autres chaînes de caractères. Le type `string` n'est pas un type élémentaire de C++. En "C" et en "C++" il existe, à la base, une notion rudimentaire de chaîne de caractères : une chaîne de caractères est un tableau (simple) de caractères se terminant par le caractère spécial `'\0'`. Par exemple, `char T[8] = "exemple" ;` définit et initialise le tableau `T` avec `T[0]= 'e'`, `T[1]='x'`, ..., `T[7]='\ 0'`. Voir (avec la commande `man`) les fonctions `strcpy`, `strcmp` et autres pour la manipulation des chaînes de caractères. Bref, il est préférable d'utiliser `string` mais sachez que d'autres solutions existent...

18 `new` et `delete`

Jusqu'à maintenant la seule manière d'allouer de la mémoire qui a été présentée, était de définir une variable ou un tableau. La gestion de ces emplacements est automatique : une variable locale est créée puis détruite à la sortie du bloc d'instructions englobant, cela correspond à la durée de vie de la variable. Il est possible d'allouer et de désallouer de la mémoire sans dépendre la durée de vie des variables : c'est le rôle des instructions `new` et `delete` :

- "`new` *type* ;" alloue un espace mémoire capable de contenir un objet de type *type* et retourne l'adresse ce cette zone mémoire qui peut être stockée dans un pointeur.
- "$P =$ `new` *type*`[`N`]` ;" alloue un espace mémoire capable de contenir N objets de type *type* et retourne l'adresse de cette zone mémoire. On parle de tableaux dynamiques que l'on manipule de la même manière que les tableaux de base (voir 17).
- "`delete` P ;" désalloue l'espace mémoire d'adresse P. Si cette zone correspond à un ensemble d'éléments (alloués par `new` *type* `[`N`]`), il faut utiliser "`delete []` P ;" sauf si *type* est un type élémentaire.

Les zones allouées par `new` ne seront désallouées que par l'appel de `delete` : la gestion de la mémoire devient votre responsabilité... gare aux fuites de mémoires !

19 structures de données dynamiques

On peut maintenant définir des structures de données dynamiques, c'est à dire dont la structure évolue au cours du temps. Il est gênant de définir une pile avec un tableau : d'une part, cela borne sa taille [12] et d'autre part une pile vide occupera autant d'espace qu'une pile pleine... Il vaut mieux allouer de l'espace mémoire au fur et à mesure des besoins : ajouter un élément à la pile correspondra à allouer (avec `new`) de la mémoire et supprimer reviendra à désallouer l'emplacement de l'objet. Le contenu de la pile sera donc éclaté dans la mémoire (car les appels successifs à `new` n'alloue pas forcément des zones contigues) et il suffit pour ne pas perdre nos éléments d'indiquer à chaque élément de pile où (l'adresse) se trouve le prochain élément (0 si il n'y en a pas). Cela correspond à la notion de *liste chaînée*. Un élément de pile d'entiers sera donc un objet de la forme :

```
struct element {
  int val;
  element * suivant;
```

[12] En fait, la classe `vector` n'impose pas cette restriction.

```
};
```

c'est-à-dire un couple comprenant un entier (la valeur à stocker dans la pile) et un pointeur sur le prochain élément de la pile. la classe pile devient alors juste une classe avec un seul champ de type `element *` correspond à l'adresse de la tête de pile. Voir le TD sur les pointeurs sur la page web.

20 Retour sur les classes

Les pointeurs sont très utilisés en C et C++. Manipuler des objets d'une classe se fait souvent au moyen de pointeur. Voici quelques informations à ce sujet :

- Si `p` désigne un pointeur sur un objet d'une classe ayant un champ `champ` et une fonction membre `fct()`, alors plutôt que d'utiliser "`(*p).champ`" ou "`(*p).fct()`", il est possible d'utiliser "`p->champ`" ou "`p->fct()`". L'opérateur "`->`" remplace donc le "`(*_).`", c'est plus court !
- Dans la définition des fonctions membres d'une classe, on utilise les champs de l'objet sur lequel les fonctions seront appelées dans le programme. Cet objet n'apparaît pas explicitement dans ces fonctions, hormis ses champs de données. Il est parfois utile de le connaître mieux et notamment de connaître son adresse mémoire : c'est possible, `this` désigne l'adresse de l'objet sur lequel la fonction définie est appelée.
- Lorsqu'on alloue de la mémoire pour stocker un objet de type `T`, un constructeur est appelé : soit celui sans argument (dans le cas d'un "`new T;`"), soit un avec argument si des arguments sont présents ("`new T(a1,...,aN);`").

D. Tout le reste

Les types génériques, les variables, champs ou fonctions `static`, les conteneurs et les itérateurs et tout le reste, vous pourrez les découvrir en cours ou dans les livres...

Remerciements. Merci à Béatrice Bérard et Gabriel Dos Reis pour leur relecture et leurs conseils.

Références

[1] A. Koenig and B. E. Moo. *Accelerated C++*. C++ In-Depth Series. Addison Wesley, 2000.

[2] S. B. Lippman. *Essential C++*. C++ In-Depth Series. Addison Wesley, 1999.

[3] S. B. Lippman and J. Lajoie. *C++ Primer*. Addison Wesley, 1998.

[4] R. Winder. *Le développement de logiciels en C++*. Masson, 1994.