Pour commencer à programmer en C++

extrait du livre : *I. Danaila, F. Hecht, O. Pironneau, Simulation numérique en C++, Dunod, 2003*

Il y a tellement de livres sur la syntaxe et la programmation en C++ qu'il serait déraisonnable de reprendre en détail ce sujet. Nous nous bornons donc ici à quelques rappels des principales notions et concepts nécessaires pour la lecture des programmes de cet ouvrage. Pour approfondir les notions exposées sommairement dans ce chapitre, il est fortement conseillé d'utiliser comme support complémentaire les livres de C++ cités en préface :

B. STROUSTRUP : *The C++ Programming Language*, Third Edition, Addison Wesley, 1997.

S. DUPIN : *Le langage C++*, Campus Press, 1999.

T. LACHAND-ROBERT : *Introduction à C++* (http://www.ann.jussieu.fr/courscpp/)

1.1 Notions de base

Le C++ étant un sur-ensemble du langage C, la plupart des notions de base exposées dans cette section sont héritées du C. Pour le lecteur déjà familiarisé avec le langage C, le survol de cette section lui permettra d'identifier les différences entre les deux langages, avant d'aborder les concepts avancés présentés dans la section suivante.

1.1.1 Fonctions et passage d'arguments aux fonctions

Le programme *ailette.cpp* (listing **??**) nous a occasionné un premier contact avec la syntaxe C++ ; nous avons constaté qu'un fichier source C++ est organisé typiquement en trois parties :

- **les directives du préprocesseur** (commençant par #).
  - le préprocesseur est appelé avant chaque compilation pour créer une image du fichier source, enrichie par le résultat de l'exécution de ces directives ; c'est finalement cette image qui sera effectivement compilée pour obtenir le fichier objet ou exécutable ;
  - par exemple, l'instruction `# include <iostream>` va rajouter dans le fichier source tout ce que le fichier système `iostream` contient, en l'occurrence les prototypes (voir plus bas) des fonctions pour les entrées/sorties ;
- **les déclarations des variables globales et des prototypes de fonctions**.
  - les variables déclarées à cet endroit seront connues dans toute fonction présente dans le fichier source, d'où leur nom de variables globales. Pour déclarer une variable, il faut préciser son type et son nom,
  - le C++ propose un grand nombre de types de base, avec la possibilité de créer ses propres

types. Les types de base les plus utilisés sont `char` (caractère), `bool` (booléen), `int` (entier), `float` (réel sur 32 bits), `double` (réel en double précision = 64 bits) ;
- le prototype (ou la signature) d'une fonction doit indiquer le type de la fonction (*i. e.* le type de la valeur de retour) et de ses arguments, pour permettre au compilateur de traiter un appel à la fonction en cause avant sa définition effective ;
- pour éviter les conflits de noms, le `C++` donne la possibilité de séparer les définitions des fonctions dans ce que l'on appelle un espace de noms (`namespace`). Si un espace de noms particulier est utilisé, il faut le préciser en début de programme par la commande `using namespace Nom ;` dans les programmes suivants l'espace standard (`std`) sera utilisé, en principal pour identifier les fonctions mathématiques usuelles et les assertions (voir plus loin) ;

- **les définitions des fonctions** constituant l'application.

Comme dans tout langage de haut niveau, la notion de fonction est à la base de la *programmation procédurale* qui permet de découper une application complexe en plusieurs modules indépendants, réutilisables dans d'autres applications. Appliquons ce concept pour transformer le programme *ailette.cpp* d'une forme monolithique, peu intéressante d'un point de vue pratique, à une forme modulaire. Mais, avant de commenter le nouveau programme, rappelons quelques règles élémentaires :

- **de syntaxe** :
  - le compilateur fait la différence entre les majuscules et les minuscules ; les mots-clés des instructions `C++` (`int, double, if, for`, etc.) s'écrivent toujours en minuscules ;
  - les espaces et les sauts de ligne sont ignorés par le compilateur, ce qui permet une écriture aérée du programme ;
  - est considéré comme commentaire, tout ce qui se trouve à droite du double slash **(//)** et tout ce qui se trouve entre **/*** et ***/**,
  - toutes les instructions doivent se terminer par un point-virgule **;**,
  - les blocs d'instructions sont délimités par des accolades $\{\cdots\}$,

- **concernant la définition des fonctions** :
  - la déclaration d'une fonction doit précéder son utilisation ;
  - le type de la fonction, ainsi que le type de ses arguments doivent toujours être déclarés ; pour appeler une fonction on doit passer des arguments qui correspondent aux types déclarés ;
  - le mot-clé `void` peut remplacer le type de la fonction pour indiquer que la fonction ne renvoie pas de valeur, ou la liste des arguments, pour indiquer que la fonction ne possède pas d'arguments ;
  - le corps de la fonction est symbolisé par le bloc d'instructions qui suit la déclaration de la fonction ;
  - une fonction ne peut renvoyer qu'une *seule valeur*, indiquée par l'instruction `return`. L'exécution de la fonction s'arrête quand cette instruction est rencontrée ;
  - le `C++` n'impose pas de récupérer la valeur renvoyée par une fonction ;
  - l'exécution de tout programme commence par la fonction `main` qui doit toujours être présente,

- **et l'utilisation des variables** :
  - le type des variables doit toujours être déclaré,
  - la déclaration d'une variable doit précéder son utilisation ,
  - une déclaration de variable peut être faite n'importe où dans un bloc d'instructions (ce qui n'est pas possible en `C`),
  - une variable peut être initialisée dès sa déclaration ;

– la portée d'une variable locale correspond au bloc où elle a été déclarée ;
– une variable globale est déclarée hors fonctions et sa portée est étendue au fichier où elle est déclarée.

Le nouveau programme *ailette2.cpp* illustre toutes ces règles (les commentaires indiquent le rôle des instructions) :

Listing 1.1 *(ailette2.cpp)*

```
                                          //   ----Directives du préprocesseur

#include <iostream>                 //    utiliser les entrées/sorties à l'écran
#include <fstream>              //    utiliser les entrées/sorties avec fichiers
#include <string>                     //    utiliser les chaînes de caractères
#include <cmath>                     //    utiliser les fonction math usuelles

using namespace std;                  //    utiliser l'espace de noms standard
//   ---------------------------------------------------------------------
//   ------Déclarations des variables ou constantes globales

                        //   ! Le type des variables doit être toujours défini

const int   MMAX=501;                 //    Définition d'une constante globale
      double L=40.,a1=4.,a2=50.;
      double To=46., Te=20.;
      double k=0.164, hc=1.e-6*200.;
      double act=2.*(a1+a2)*hc*L*L/(k*a1*a2);

//   ---------------------------------------------------------------------
                     //   ------Définition des fonctions
   //   ! La définition des fonctions doit précéder leur utilisation

//   Fonction sans aucun argument (autre déf possible: int Compteur( void) )
//   ========================================================================
int Compteur()           //    Le type des fonctions doit être toujours défini
{                                          //   ---------------------début bloc
  static int i=0;         //    Var statique(sa valeur est gardée d'un appel a
l'autre)
                    //    La variable peut être initialisée dès sa déclaration
  i++;                             //    La var i est incrémentée de 1 (i=i+1)
                         //    Toute instruction dans un bloc doit finir avec ;
  return i;              //    Valeur (une seule!!!) retournée par la fonction
}                                          //   ----------------------fin bloc

//   Fonction simple avec argument de type double
//   ============================================================================
double Solexacte(double x)
{
    return  (To-Te)/Te*cosh(sqrt(act)*(1-x))/cosh(sqrt(act));
}

//   Fonctions avec arguments de type:
//        pointeur-tableau et pointeur-chaîne de caractères
```

```
//   ========================================================================
int AfficheTab(int M, double *pTab1, double *pTab2, char *Message)
{
                        //    Affiche à l'écran les vecteurs p*Tab en utilisant:
               //    le flux de sortie, "endl" ou "\n" indique un saut de ligne
                                          //    "\t" introduit une tabulation
      cout << Message <<endl;
      cout << "Dimension du vecteur  = "<< M <<endl;

                                                  //    Exemple de boucle for
  for(int i=0;i<M;i++)
  {                                                 //   -----début bloc for
              //    Accès à la valeur de l'élément du tableau par son pointeur
   cout<< "Element "<< i <<"\t"<<pTab1[i]<<"\t"<<pTab2[i]<<"\n";
  }                                                  //   -----fin bloc for

 return 0; //    Valeur retournée (superflue) => void AfficheTab(...) possible
}


void SauveTab(int M, double *pTab1,  double *pTab2,
              double *pTab3, char *NomFic)
{
                      //    Sauvegarde des vecteurs pTab dans le fichier NomFic

    ofstream fic(NomFic);                          //    on ouvre le fichier
     for(int m = 0; m< M; m++)        //    boucle for à une seule instruction
       fic<<pTab1[m]<<"\t"<<pTab2[m]<<"\t"<<pTab3[m]<<endl;
    fic.close();                                   //    on ferme le fichier
}

//   Fonctions avec arguments: pointeur-tableau et pointeur-fonction
//   ========================================================================
void VectExacte(int M, double* pTab, double* pX, double (*pFunc)(double x))
{
    for(int m=0;m<=M;m++)
    {
                           //    Appel de la fonction en utilisant son pointeur
    pTab[m]=pFunc(pX[m]);
    }
}

//   Les autres fonctions
//   ========================================================================
void Abscisses(int & M, double * pX)
{
    double h=1./M;
    for(int m=0;m <= M;m++)
         pX[m]=m*h;
}


void InitSol(int * M, double* pX, double* pTheta)
{
     pTheta[0]=(To-Te)/Te;
     for(int m=1;m <= (*M);m++)
```

```
        pTheta[m]=(1.-pX[m])*(To-Te)/Te;
}

void GaussSeidel(int M, int itermax, double prec, double* pTheta)
{
 double h=1./M;double epsilon=0.;
     do
       { epsilon=0.;double xnew;
         for(int m=1;m < M;m++)
             { xnew  = (pTheta[m-1]+pTheta[m+1])/(2.+h*h*act);
               epsilon += (xnew-pTheta[m])*(xnew-pTheta[m]);
               pTheta[m] = xnew;
              }

           xnew=pTheta[M-1];                              //   condition en x=1
           epsilon += (xnew-pTheta[M])*(xnew-pTheta[M]);
           pTheta[M]=xnew;

       }while((sqrt(epsilon) > prec) && (Compteur() < itermax) );


    int iter=Compteur();
    if(iter<itermax)
      cout << "M="<<M<<"  Convergence en GS aprés "<<iter<<" iterations\n";
    else
      cout << "Stop en GS aprés "<<iter<<" iterations\n";
}

//   Fonction principale ou main (l'exécution commence par main)
//   ===========================================================================
void main( )
{
                                                  //    Construction du maillage
  int M;
     cout << "Dimension du maillage (N<500)"<<endl;
     cin >> M;
             //    Définition d'un tableau de taille fixe (création statique);
                                     //    sa dimension doit être une constante
  double xa[MMAX];
  Abscisses(M,xa);

                                    //    Initialisation (distribution linéaire)
                              //    Définition d'un tableau dynamique (pointeur);
                                         //    seulement M éléments sont alloués

  double * theta=new double[M+1];
  InitSol(& M,xa,theta);

                                                //    Résolution par Gauss-Seidel
  GaussSeidel(M,100000,1e-8,theta);

                                               //    Solution (exacte) analytique
   double * thetaa= new double[M+1];
   VectExacte(M,thetaa,xa,Solexacte);

                           //    Affichage à l'écran (valeurs adimensionnalisées)
```

```
   AfficheTab(M+1,theta,thetaa,"Sol num / Sol exacte");

                      //    Sauvegarde du résultat dans un fichier pour gnuplot
    SauveTab(M+1,xa,theta,thetaa,"result.dat");
}
```

Analysons maintenant le programme d'un point de plus pratique, celui du **passage d'arguments aux fonctions**. Trois techniques peuvent être utilisées en C++ :

- **passage par valeur** :
  au niveau de la fonction, la variable argument est initialisée par une copie d'une variable de la fonction appelante. C'est le cas de l'argument M de la fonction AfficheTab : la valeur de la variable M de la fonction `main` (fonction appelante) est copiée dans une variable locale, du même nom M, qui sera détruite après l'exécution de la fonction. La variable passée comme argument ne peut pas être modifiée dans ce cas.
- **passage par pointeur** :
  l'adresse mémoire (pointeur, noté *** pvar**) d'une variable de la fonction appelante est passée à la fonction. Cette méthode permet de modifier directement les valeurs des plusieurs variables de la fonction appelante, contournant ainsi la restriction imposant une seule valeur de retour pour une fonction. En particulier, c'est le seul moyen de passer un tableau à une fonction (voir plus loin).
- **passage par référence** :
  c'est une solution hybride entre le passage par valeur et par pointeur. La fonction appelée utilise un synonyme (référence, notée **& rvar**) de la variable argument. C'est bien la variable, et non son adresse qui est passée à la fonction, mais ce type d'appel permet de modifier la valeur de la variable définie dans la fonction appelante. C'est le cas de l'argument `M` de la fonction `Abscisses` ; même si l'exemple est un peu forcé, car nous n'avons pas besoin de modifier le valeur de `M`, il montre qu'à l'intérieur de la fonction on utilise cet argument comme s'il s'agissait d'une variable classique. L'appel de la fonction, à partir de `main`, se fait en utilisant le nom d'une variable de type `int`.

Remarque 1.1 *La référence peut être vue comme un pointeur constant pour lequel on prend automatiquement la valeur pointée. Comme l'adresse mémoire de la variable référée ne peut pas être modifiée, l'utilisation d'une référence revient à donner un autre nom à une zone mémoire de la machine.*
Une référence doit toujours être initialisée !

1.1.2 Pointeurs et tableaux

Plusieurs **types de pointeurs** sont utilisés dans le programme *ailette2.cpp* : pointeur-tableau, pointeur-caractère et pointeur-fonction. Quelques précisions sur cette notion très importante (et difficilement maîtrisable par le programmeur débutant) s'imposent :

– le pointeur est une variable (**pVar**) qui contient une double information concernant une autre variable (**Var**) : l'adresse mémoire du premier mot mémoire (ou octet=8 bits) où la variable Var est stockée et combien de mots mémoire (**nMots**) sont utilisés. Comme la valeur de (nMots) dépend du type de la variable, le type d'un pointeur (le même que celui de la variable associée)

doit toujours être déclaré ;

– une fois déclaré, un pointeur doit être initialisé. Comme syntaxe, **&Var** retourne l'adresse mémoire de la variable Var et ***pVar** désigne la valeur stockée à l'adresse représentée par le pointeur pVar. Par exemple, si nous définissons une variable et un pointeur de type `double`
`double Var ; double * pVar ;`
le pointeur peut être initialisé par l'adresse mémoire de `Var` en écrivant
`pVar = & Var ;`
Dans ce cas, nous avons deux possibilités pour modifier la valeur de la variable `Var` :
`Var=valeur ;` (classique), ou bien
`*pVar=valeur ;` qui permet de travailler directement avec l'adresse mémoire de la variable ;

– **arithmétique des pointeurs** :
`pVar=pVar+1 ;` fait que pVar pointe vers une nouvelle case mémoire, obtenue par le déplacement d'un nombre de mots mémoire (nMots). Par conséquent, pVar va stocker l'adresse de la variable suivante.

Remarque 1.2 *Les pointeurs constituent un outil très puissant pour la programmation qui doivent être utilisés avec beaucoup de rigueur ; il est très facile d'écraser involontairement des zones mémoire du programme ou de récupérer des valeurs complètement fausses.*

Un exemple de passage d'argument par pointeur est donné dans l'écriture de la fonction `InitSol`. Il s'agit de l'argument `M` qui est un pointeur de type `int` : à l'intérieur de la fonction nous avons besoin de la valeur correspondant à cette adresse mémoire, donc nous utilisons (`*M`) pour la désigner. L'appel de cette fonction est fait dans `main` en envoyant l'adresse mémoire (`&M`) d'une variable du même type `int`.

Dans le calcul numérique la notion de **tableau** joue un rôle crucial et le `C++` offre une grande flexibilité dans l'utilisation de cette structure de données. En `C++` le nom d'un tableau désigne l'adresse mémoire de son premier élément (*i.e.* l'élément 0) et correspond de ce fait à un pointeur. Par conséquent, un tableau est complètement défini par son pointeur et sa taille. Tenant compte de l'arithmétique des pointeurs , il y a une parfaite équivalence entre la notation tableau et pointeur. Par exemple, si `Tab` est un tableau de dimension 20 et `pTab` un pointeur qui pointe vers `Tab`,
`double Tab[20] ;double *pTab ;`
`pTab=Tab ;` $\Longleftrightarrow$ `pTab=&Tab[0] ;`
nous avons équivalence des instructions suivantes :
`Tab[0]=1. ;` $\Longleftrightarrow$ `pTab[0]=1. ;` $\Longleftrightarrow$ `*Tab=1. ;`
`Tab[1]=2. ;` $\Longleftrightarrow$ `pTab[1]=2. ;` $\Longleftrightarrow$ `*(Tab+1)=2. ;`

En pratique, on utilise la notation tableau (beaucoup plus simple et intuitive) dans tous les cas. L'équivalence pointeur $\leftrightarrow$ tableau nous a donné la possibilité de définir dans la fonction `main` des :

– **tableaux de taille fixe (création statique) :** `double xa[MMAX] ;`
la dimension (`MMAX`) doit être une constante pour que le compilateur puisse réserver les cases mémoire nécessaires, ce qui ne permet pas d'utiliser la mémoire de manière optimale (en réalité

on utilise seulement `M+1` éléments de ce tableau) ; ce type de tableau est à utiliser avec parcimonie car la mémoire disponible pour les variables locales est petite, pour les tableaux de taille importante (> 1000), utilisez les tableaux dynamiques ;

– **tableaux dynamiques :** `double * theta=new double[M] ;`
il s'agit d'un pointeur-tableau de dimension `M` - l'espace mémoire nécessaire est alloué de manière dynamique au moment de l'exécution du programme et non pas à la compilation comme pour les tableaux statiques. Ceci permet de réserver seulement la place mémoire nécessaire aux calculs.

Remarque 1.3 *Les tableaux dynamiques doivent être manipulés avec attention car les dépassements de tableaux ne peuvent pas être signalés au moment de l'exécution du programme. Un dépassement de tableau dynamique va accéder naturellement à la case mémoire qui suit celles qui stockent le tableau et peut entraîner l'écrasement d'une autre variable du programme.*

Remarque 1.4 *L'utilisation d'un pointeur-tableau est une possibilité pou passer un tableau comme argument d'une fonction. Cette technique permet de modifier les éléments d'un tableau défini dans une autre fonction (voir les fonctions `Abscisses, InitSol`, etc.).*

Un tableau particulier intervient en `C++` dans la définition d'une **chaîne de caractères**, considérée comme un tableau de type caractère **char** (il n'existe pas un type chaîne de caractères). Par conséquent, le pointeur-chaîne de caractères est un pointeur de type tableau. Toute chaîne de caractères finit avec le caractère spécial « `\0` », donc la taille de la chaîne est automatiquement déterminée une fois que son pointeur est connu. Ce type de pointeurs a été utilisé dans la fonction `SauveTab` pour passer un nom de fichier et dans la fonction `AfficheTab` pour passer un message.

Un dernier type de pointer utilisé dans le programme est le **pointeur de fonction** : il contient l'adresse de la zone mémoire qui stocke le code binaire de la fonction. Ce type de pointeur est utilisé dans la fonction `VectExacte`). Remarquons que dans la définition du pointeur le type de la fonction acceptée et le type de ses arguments sont indiqués. En conséquence, l'appel de la fonction (en `main`) doit comporter comme argument le nom d'une fonction (en l'occurrence `Solexacte`) dont le prototype correspond à la définition du pointeur.

1.1.3 Synthèse des déclarations `C++`

À partir des types de base (`char`, `short`, `bool`, `int`, `long`, `long long`, `float`, `double`) le `C++` offre une grande liberté dans la définition des variables, pointeurs, références, tableaux et des fonctions sur ces types. Le tout nous donne une algèbre des types qui est loin d'être triviale.

Nous présentons dans le tableau 1.1 les principales déclarations `C++` qui peuvent être construites à partir des types de base `T`,`U`.

Remarque 1.5 *Il n'est pas possible de construire un tableau de références car il serait impossible de l'initialiser.*

Remarque 1.6

Une fois définie, une structure dynamique (pointeur) doit être initialisée par allocation de l'espace mémoire nécessaire. La syntaxe déjà utilisée pour allouer un pointeur -tableau

```
T * a = new  T[10]
```

se généralise, par exemple, pour l'allocation d'un tableau de pointeurs de fonctions `U →T`

```
T (**a)(U) = new (T (*[10])(U)) ; .
```

déclaration	prototype	description du type
`T * a`	`T *`	un pointeur sur une variable (objet) de type `T`
`T a[10]`	`T[10]`	un tableau de `T` composé de 10 variables de type T
`T a(U)`	`T a(U)`	une fonction d'argument `U` qui retourne un `T` (`U →T`)
`T &a`	`T &`	une référence sur un objet de type `T`
`const` `T a`	**`const`** `T`	un objet constant de type `T`
`T` **`const`** `* a`	`T` **`const`** `*`	un pointeur sur un objet constant de type `T` [a]
`T *` **`const`** `a`	`T *` **`const`**	un pointeur constant sur objet de type `T` [b]
`T` **`const`** `*` **`const`** `a`	`T` **`const`** `*` **`const`**	un pointeur constant sur objet constant
`T * & a`	`T * &`	une référence sur un pointeur sur `T`
`T ** a`	`T **`	un pointeur sur un pointeur sur `T`
`T * a[10]`	`T *[10]`	un tableau de 10 pointeurs de type `T`
`T (* a)[10]`	`T (*)[10]`	un pointeur sur tableau de 10 éléments de type `T`
`T (* a)(U)`	`T (*)(U)`	un pointeur sur une fonction `U →T`
`T (* a[])(U)`	`T (*[])(U)`	un tableau de pointeurs sur des fonctions `U→T`

[a] Les instructions qui font intervenir le mot-clé `const` doivent être lues de droite à gauche. Dans ce cas, `a` sera un pointeur vers un `const T`.

[b] Il s'agit ici d'un `const` pointeur sur un `T`.

Tableau 1.1 – Déclarations possibles en C++ .

1.1.4 Structures de contrôle

Une dernière notion de base nécessaire pour écrire des programmes C++ est constituée par les structures de contrôle. Les tests et les boucles C++ utilisent la syntaxe donnée dans le tableau 1.2.

Remarque 1.7

Un expression utilisée comme condition logique est vraie si elle est non nulle. Par conséquent, il y a parfaite équivalence entre les instructions suivantes :

`if` `(x != 0)` *et* **`if`** `(x)`,

`if` `(x == 0)` *et* **`if`** `(!x)`.

1.1.5 Exercices

Exercice 1.1

Ajouter au programme *ailette2.cpp* une fonction qui résout un système linéaire à matrice tridiagonale par une méthode directe (factorisation LU). Utiliser cette fonction pour le calcul de la distribution de température dans l'ailette.

if (condition)
{ //instructions 1
}
else
{ //instructions 2
}
	avec	condition = (cond1) **op-logique** (cond2) ...
		op -logique ∈ { &&(et), || (ou), ! (négation)}
		cond1= (expr1) **op-relation** (expr2)
		op-relation ∈ { > , < , <= , >= , == , != }

Opérateur ternaire z=(cond) ? var1 : var2 ;	⟺	**if** (cond) z=var1 ; **else** z=var2 ;

for (exp1 ; exp2 ;
exp3)
{ //instructions
}
	avec	exp1 = déclare et initialise la variable compteur
		exp2 = impose la condition de sortie (tant que ...)
		exp3 = incrémente le compteur

while (condition)
{ //instructions
}
	ou	**do**
		{ //instructions
		}**while**(condition) ;

Tableau 1.2 – Structures de contrôle en C++ .

Exercice 1.2

Récrire le programme *ailette2.cpp* pour un maillage variable décrit par les abscisses des points de discrétisation :

$$x_m = L \cdot \sin(\varepsilon \frac{\pi}{2} \frac{m}{M}) / \sin(\varepsilon \frac{\pi}{2}), \quad i = 0, 1, \ldots, M, \quad \text{avec} \quad \varepsilon = 3/4.$$

(Attention, la discrétisation de l'opérateur différentiel doit tenir compte de cette distribution des points du maillage !)

1.2 Classes et objets

Le programme analysé dans la section précédente illustre la technique de programmation de base du langage C, qui est l'approche procédurale (basée sur l'utilisation des fonctions spécialisées). Le C++ a marqué une évolution majeure par rapport au C en introduisant des facilités de programmation orientée objet (POO). L'idée de la POO est de pouvoir construire des applications spécialisées à partir de quelques structures de base, assez abstraites pour être utilisées dans des contextes tout à fait différents. Par conséquent, l'effort le plus important dans la POO est situé au niveau de la conception des structures (`classes`) de base[1], contrairement à la programmation classique qui

[1]Des classes *performantes*, mais complexes, font généralement l'objet des bibliothèques permettant à l'utilisateur de développer ses propres applications. Le lecteur peut se constituer une vraie bibliothèque pour le calcul scientifique

se concentre sur la construction des fonctions spécialisées. Pour obtenir un maximum de flexibilité dans l'utilisation de ses outils de base, le programmeur `C++` dispose de plusieurs techniques, présentées rapidement dans cette section.

1.2.1 Définition des classes

La notion d'**objet** permet de grouper en une seule entité des variables et des fonctions (technique appelée parfois encapsulation). Les objets sont générés suivant la structure définie par la déclaration d'une **classe**. Autrement dit, une classe représente le moule qui permet de créer des objets. Le concept de classe est en fait une extension de la notion de type ; à part les types classiques (hérités du C), le `C++` offre au programmeur la possibilité de définir des types plus complexes, en groupant des définitions de variables et de fonctions.

Dans la définition d'une classe, plusieurs éléments peuvent être présents :

• les **données membres**, constituées de variables de type connu.

• les fonctions membres ou **méthodes**. Il est important de savoir que l'ensemble des données membres est accessible par le pointeur constant **this**, qui est passé par le compilateur comme argument caché dans toute fonction membre. Par conséquent, dans l'écriture d'une méthode, nous pouvons appeler directement une variable membre, soit en utilisant naturellement son nom, disons `var1`, soit par la syntaxe `this->var1`. Cette dernière écriture permet au programmeur d'éviter la confusion avec des variables locales qui portent le même nom. Nous allons voir plus tard une utilisation explicite plus intéressante du pointeur `this` pour la manipulation des listes chaînées.

Une technique très intéressante pour la définition des méthodes est la **surcharge de fonctions**, qui permet de définir plusieurs fonctions portant le même nom, mais avec des arguments différents. Au moment de l'exécution, le compilateur va vérifier le type arguments envoyés et va sélectionner la fonction correspondante. Il faut retenir que la valeur de retour n'est pas vérifiée, elle ne peut donc pas jouer dans la surcharge de fonctions.

Remarque 1.8 *Il est impossible de surcharger une fonction en utilisant un appel par valeur et un autre par référence car le compilateur ne peut pas distinguer entre les deux au moment de l'appel.*

• le **constructeur** est une méthode avec une définition particulière : elle porte le nom de la classe et la valeur de retour ne doit pas être indiquée. Comme toute méthode, le constructeur peut être surchargé, donc on peut disposer de plusieurs constructeurs dans une même classe. Un constructeur sans arguments (*par défaut*) est intégré automatiquement par le compilateur à tout classe qui ne possède aucun constructeur. Le rôle principal des constructeurs est d'initialiser les données membres.

Remarque 1.9 *Une fois un constructeur défini, le compilateur ne génère plus le constructeur par défaut, donc il va falloir le définir si on compte l'utiliser.*

• les **opérateurs** sont des fonctions spéciales qui permettent d'effectuer des opérations avec les données abstraites de la classe ; l'utilisateur peut redéfinir pour ses données les opérateurs `C++`

en suivant les applications de cet ouvrage.

+	-	*	/	%	^	&	|	~	!	=	<	>	new[]
-=	*=	/=	%=	^=	&=	|=	<<	>>	<<=	>>=	==	!=	delete
>=	&&	||	++	--	->*	,	->	[]	()	new	+=	<=	delete[]

en utilisant la syntaxe suivante pour un opérateur avec `n` arguments (opérateur unaire si `n=1` ou binaire si `n=2`) :

```
Type-retour operator  symbole  (arg1, ..., argn)
```

Remarque 1.10 *Les opérateurs peuvent être définis, comme toute fonction, indépendamment des classes. Quand ils sont définis dans une classe, ils sont considérés comme méthodes et le pointeur `this` devient implicitement le premier argument (caché) de l'opérateur. Par conséquent, il est nécessaire de spécifier seulement `n-1` arguments dans sa définition.*

• le **destructeur**, comme son nom l'indique, est une méthode qui sera appelée implicitement pour détruire les objets et libérer la place mémoire (voir plus loin). Comme méthode, le destructeur est facile à identifier car il porte le nom de la classe précédé par le symbole « ~ » et ne possède pas d'argument. Ne pouvant pas être surchargé, le destructeur est unique dans une classe.

Remarque 1.11 *Il faut savoir qu'une variable locale est détruite à la sortie du bloc contenant sa définition par un appel automatique du destructeur. En conséquence, le programmeur n'a généralement pas besoin d'appeler explicitement un destructeur.*

1.2.2 Un premier exemple : la classe vide

Pour mettre en évidence le rôle des constructeurs et des opérateurs, considérons l'exemple d'une classe vide, qui ne possède ni données, ni méthodes. Le programme suivant construit cette classe et suit la logique d'appel des composantes de la classe au moment de la création des objets.

Listing 1.2 *(ClassVide.cpp)*

```
#include <iostream>
using namespace std;

class Vide
{public:
     Vide()
     {cout << "Constructeur par défaut  à l'adresse  "<< this << endl;}
     Vide(const Vide & t)
     {cout << "Constructeur par copie   à l'adresse  "<< this <<endl;}
     ~Vide()
     {cout << "Destructeur  de l'objet    d'adresse  "<< this <<endl;}

Vide & operator = (Vide &t)
        {cout << "Operateur = pour Vide"<< endl;
         cout << "  on copie l'objet d'adresse          "<< &t << endl;
```

```
          return t;}
};

Vide  & fct1(Vide v)
{   return v;}

Vide &  fct2(Vide &v)
{   return v;}

void main()
{  Vide a, b(a);
        cout<<"--------------------"<< endl;
   b=fct1(a);
        cout<<"--------------------"<< endl;
   b=fct2(a);
        cout<<"--------------------"<< endl;
   Vide * c = new Vide;
   Vide * d = new Vide(a);
        cout << "c pointe vers l'adresse  "<< c << endl;
        cout<<"--------------------"<< endl;
        c=d;
        cout << "c pointe vers l'adresse  "<< c << endl;
        cout<<"--------------------"<< endl;
   delete c;
        cout << "c pointe vers l'adresse  "<< c << endl;
        cout<<"--------------------"<< endl;
}
```

Remarquons d'abord quelques éléments de syntaxe :

• la ligne de définition comporte le mot-clé `class` suivi du nom de la classe ;

• le bloc de définition de la classe finit toujours avec un point-virgule ;

• le niveau de protection des composantes de la classe est indiqué par l'étiquette qui précède leur définition. Il existe trois niveaux de protection[2] qui peuvent être présents dans une même classe et séparer la définition de ses composantes :
– `public` (*public*) : utilisation libre des données et méthodes par toutes les fonctions du programme ;
– `private` (*privé*) : les données et les méthodes sont accessibles seulement par les fonctions de la classe ;
– `protected` (*protégé*) : les composantes ainsi définies sont utilisables par les fonctions de la classe et des classes dérivées (voir la section suivante).

Remarque 1.12 *Les éléments de la classe définis sans aucune étiquette de protection sont considérés de type* `private`*.*

Analysons maintenant la structure de la classe :

[2] Voir également le paragraphe **??** sur les fonctions et les classes amies.

- elle contient deux constructeurs : le premier est sans argument (constructeur par défaut) et ne fait qu'afficher le pointer `this` qui contient l'adresse mémoire (nombre hexadécimal) attribué à la création d'un nouveau objet ; le deuxième constructeur va créer un objet en copiant celui correspondant à la référence passée en argument (en réalité, la classe étant vide, ce constructeur par copie ne fait rien) ;

- le destructeur affiche l'adresse mémoire de l'objet détruit ;

- l'opérateur « = » s'applique à une référence de type `Vide` et retourne cette référence, donc réalise l'opération d'affectation habituelle.

Remarque 1.13 *Le constructeur par copie et l'opérateur d'affectation « = » ont des rôles tout à fait différents : le premier initialise de la mémoire pour un objet non-construit, alors que le deuxième doit traiter un objet parfaitement construit.*

1.2.3 Création d'objets

Au vu de la syntaxe utilisée dans la fonction `main`, la création d'objets semble naturelle, similaire à la déclaration des variables classiques. Effectivement, une classe n'étant que la définition d'un nouveau type, toutes les déclarations du tableau (1.1) sont possibles, en remplaçant `T` par le nom de la classe. Il va de même pour le passage d'arguments aux fonctions discuté dans le paragraphe 1.1.1. Il existe néanmoins quelques différences :

- la création statique d'un objet local (`a` et `b` dans le programme) implique un appel implicite au constructeur de la classe. Pour sélectionner un constructeur particulier, il faut spécifier les arguments appropriés : si aucun argument n'est indiqué, le constructeur par défaut (qui existe toujours) est appelé (c'est le cas de l'objet `a`) ; une fois `a` créé, il peut être copié pour générer l'objet `b`, en utilisant le constructeur par copie.

- les objets dynamiques (pointeurs), `c` et `d` dans le programme, ont besoin d'être initialisés et la manière la plus simple de le faire et d'utiliser l'opérateur `new`. Cet opérateur (déjà utilisé pour la création de tableaux dynamiques) va allouer dynamiquement la mémoire nécessaire en appelant implicitement les constructeurs appropriés (le constructeur par défaut pour `c` et le constructeur par copie pour `d`).

- un objet local est détruit lorsque l'on quitte son bloc de définition, tandis qu'un objet dynamique créé par `new` doit être explicitement détruit par l'opérateur `delete`. Dans les deux cas, le destructeur est appelé implicitement pour libérer l'espace mémoire occupé par les objets détruits.

Remarque 1.14 *L'opérateur `delete` ne peut être appliqué qu'aux pointeurs initialisés par `new`.*

La suite logique d'appel des constructeurs, opérateurs et destructeurs de la classe peut être suivie facilement grâce aux messages affichés à l'exécution du programme :

```
Constructeur par défaut  à l'adresse  0x75fd1c                    //    Vide a
Constructeur par copie   à l'adresse  0x75fcfc                  //    Vide b(a)
--------------------
Constructeur par copie   à l'adresse  0x75fcdc                    //    fct1(a)
Destructeur  de l'objet    d'adresse  0x75fcdc
```

```
Opérateur = pour Vide                                          //    b=fct1(a)
  on copie l'objet d'adresse          0x75fcdc
--------------------
Opérateur = pour Vide                                          //    b=fct2(a)
  on copie l'objet d'adresse          0x75fd1c
--------------------
Constructeur paré  à l'adresse  0x9906d8                     //    c=new Vide
Constructeur par copie   à l'adresse  0x9906e8             //    d=new Vide(a)
c pointe vers l'adresse  0x9906d8
--------------------
c pointe vers l'adresse  0x9906e8
--------------------
Destructeur  de l'objet    d'adresse  0x9906e8                  //    delete c
c pointe vers l'adresse  0x9906e8
--------------------
Destructeur  de l'objet    d'adresse  0x75fcfc
Destructeur  de l'objet    d'adresse  0x75fd1c
```

Nous avons mis en parallèle les instructions qui ont généré chaque groupe de messages. Analysons rapidement ces messages.

- Les deux premières lignes nous indiquent les adresses mémoire des objets créés avec le constructeur par défaut (`a`) et par copie (`b`).

- L'appel de la fonction `fct1` utilise un passage d'argument par valeur, donc il est nécessaire de copier l'argument dans une variable locale (constructeur par copie) qui porte le même nom `v`. L'opérateur « = » sera ensuite utilisé pour copier en `b` le résultat envoyé par `fct1` ... sauf que, et les messages l'indiquent clairement, le contenu de la zone mémoire attribuée à la variable locale `v` a été effacé après l'exécution de la fonction ! Conclusion de cette analyse : une fonction ne doit jamais renvoyer la référence (ou l'adresse) d'une variable locale, car elle est détruite en quittant la fonction.

- La situation est différente pour `fct2` qui prend comme argument une référence : la copie n'est plus nécessaire et la valeur de `a` est attribuée directement à `b` par l'opérateur « = ».

- L'opérateur `new` fait implicitement appel aux constructeurs (par défaut pour `c` et par copie pour `d`). Il est intéressant de remarquer ici que l'attribution $c = d$ utilise, comme attendu, l'opérateur « = » défini pour les nombres hexadécimaux.

- L'instruction `delete c` implique l'effacement de la zone mémoire pointée par le pointeur `c` ; le pointeur, quant à lui, existe toujours et reste disponible pour une nouvelle affectation (par l'opérateur `new`, par exemple).

- À la fin du programme, les objets statiques `a,b` sont détruits automatiquement ; ce n'est pas le cas du pointeur `d` qui aurait dû être détruit par l'opérateur `delete`.

Exercice 1.3

Modifier l'opérateur « = » pour retourner un objet de type `Vide` à la place d'une référence `Vide &`. Expliquer les nouveaux messages obtenus.

Remarque 1.15

La classe `Vide` ne contenant ni données, ni méthodes, nous n'avons pas eu l'occasion d'accéder les éléments d'une classe. La syntaxe générale, qui sera utilisée par la suite, est la suivante : si `T` est une classe contenant, par exemple, une donnée `var1` et une méthode `meth1()` sans arguments, alors :
- un objet statique/dynamique est déclaré par

`T obj;`	`T * pobj;`

- et ses composantes sont accessibles en utilisant respectivement les opérateurs "." (point) et "−>" (flèche = moins + plus grand)

`obj.var1;`	`pobj -> var1;`	`(*pobj).var1;`
`obj.meth1();`	`pobj -> meth1();`	`(*pobj).meth1();`

1.3 Programmation générique (templates)

Nous avons vu que la correspondance des types pour l'utilisation des fonctions ou des classes est rigoureusement vérifiée par le compilateur. Afin d'éviter d'écrire plusieurs fonctions (ou classes) qui appliquent le même traitement à plusieurs types de données, le C++ offre la possibilité d'utiliser les `templates` (ou modèles). Le type de données devient un paramètre dans la définition des classes ou des fonctions et il sera choisi au moment de leur utilisation. On parle alors de *types paramétrés*.

1.3.1 Fonctions templates

Un exemple simple d'utilisation des `templates` est donné dans le programme *ClassComplex.cpp*. La fonction `Max` est définie avec un type paramétré `T` qui sera le type de la valeur de retour et celui des arguments[3]. Ce type est indiqué au moment de l'appel de la fonction, dans `main` : pour le premier appel `T=int` et pour le deuxième `T=double`. Remarquons que l'appel `Max(a,c)` sera interdit par le compilateur, car la définition de `Max` comporte un seul type paramétré et `a` et `c` sont de types différents.

Listing 1.3 *(ClassComplex.cpp)*

```
#include <iostream>
#include <cmath>
using namespace std;

typedef double Reel;

template <class T>
T Max(const T & a, const T & b){T z=a>b? a : b; return z;}

class Complex
{public:
    Reel x,y;
    Reel Norme() const;
```

[3]La fonction `Max` utilise l'opérateur ternaire défini dans le tableau 1.2.

```
    Complex( ): x(0), y(0){ }
    Complex(Reel a, Reel b);
};

Complex::Complex(Reel a, Reel b)
{   x=a;y=b; }

Reel Complex::Norme() const
{   return sqrt(x*x+y*y); }

bool operator > (const Complex & a, const Complex & b)
{   return a.Norme() > b.Norme()? true : false;  }

ostream & operator << (ostream & f, const Complex & a)
{   f<<"("<<a.x<<","<<a.y<<")"; return f;  }

void main()
{   int a=3, b=5;
    cout <<"Max de "<<a<<"  et  "<< b<< "  est  " <<Max(a,b)<<endl;
    double c=1.5, d=2.5;
    cout <<"Max de "<<c<<"  et  "<< d<< "  est  " <<Max(c,d)<<endl;
    Complex f(1,2), g(1.2,3.2);
    cout <<"Max de "<<f<<"  et  "<< g<< "  est  " <<Max(f,g)<<endl;
}
```

Ce qui est intéressant dans cet exemple est que la fonction `Max` peut être utilisée avec des types plus abstraits, une fois qu'une relation d'ordre est définie. Autrement dit, on peut comparer les objets appartenant à une classe si l'opérateur « `>` »est défini. C'est le cas de la classe `Complex` dans ce programme. Cette classe construit un nombre complexe, à partir de deux réels et compare leurs normes euclidiennes. Si au niveau conceptuel cette classe est vraiment très simple, elle introduit quelques éléments nouveaux de syntaxe :

- L'instruction `typedef` est utilisée pour définir un alias d'un type connu. Dans notre cas le type `Reel` est synonyme du type `double`. Cette possibilité de définir ses propres noms pour les types de données facilite la lisibilité et la généralité des programmes.

- Le constructeur par défaut de la classe est défini à l'intérieur de la classe
`Complex( ) : x(0), y(0) { }`
et utilise *une liste d'initialisation* débutant par le symbole « `:` » Il s'agit d'une manière plus compacte d'écrire
`Complex( ) {x=0;y=0;}`

- Le deuxième constructeur et la méthode `Norme` sont définis à l'extérieur de la classe (*définition déportée*). Dans ce cas, il faut indiquer leurs prototypes à l'intérieur de la classe et dans la ligne de définition, leur nom doit être précédé par le nom de la classe suivi de l'opérateur « `::` » qui est opérateur de résolution de portée (noté ORP). Cette syntaxe indique clairement au compilateur l'appartenance d'une fonction à une classe.

- le mot-clé `const` est utilisé à deux reprises, avec des significations différentes. À la fin de la définition d'une méthode, `const` indique au compilateur que la méthode ne peut pas modifier les

données membres de la classe (c'est le cas de la méthode `Norme`). Cette technique de programmation assure un niveau de protection supplémentaire contre la modification accidentelle des données membres. Le mot-clé `const` est également utilisé dans la définition du type des arguments de l'opérateur « > » ; il indique cette fois que les objets dont les références sont passées comme arguments ne peuvent pas être modifiés. Il s'agit, de nouveau, d'un moyen simple de protection des objets intervenant dans les calculs effectués par certaines fonctions.

- Les opérateurs « > » et « << » sont redéfinis indépendamment de la classe, comme des opérateurs binaires. L'opérateur « > » compare les normes des objets et retourne un booléen (remarquons l'écriture compacte à l'aide de l'opérateur ternaire) qui permettra à la fonction `Max` d'évaluer la condition `a>b`. L'opérateur de flux de sortie « << » est redéfini afin d'afficher un `Complex` comme un couple de deux réels ; il retourne une référence vers un objet de type `ostream` (type défini dans le fichier en-tête `iostream`) qui est utilisée par défaut par la fonction `cout`.

Comme attendu, le résultat de l'exécution du programme est le suivant :

```
Max de 3  et  5  est  5
Max de 1.5  et  2.5  est  2.5
Max de (1,2)  et  (1.2,3.2)  est  (1.2,3.2)
```

1.3.2 Classes templates

Les `templates` peuvent être également utilisés pour paramétrer les types des données membres. Les classes templates deviennent intéressantes quand il est nécessaire de gérer des collections d'objets, comme les tableaux. Le programme suivant construit une classe tableau avec des éléments de type `T`, défini comme paramètre.

Listing 1.4 *(ClassVect.cpp)*

```
#include <iostream>
#include <cassert>
using namespace std;

template <class T>
class Vect
{public:
    int n;
    T *v;
    void Affiche(char* Message) const;

    Vect(int nn): n(nn), v(new T[n]){assert(v || !n);}
    Vect(const Vect & w);
   ~Vect(){if(v) delete [] v;}

    void    operator = (const Vect & a);
    T & operator [] (int i) {assert(v && i>=0 && i < n);return v[i];}
};

template <class T>
```

```
Vect<T>::Vect(const Vect &w):n(w.n),v(new T[n])
{   (*this)=w;  }

template <class T>
void Vect<T>::operator = (const Vect & a)
{   assert(!n || (n==a.n && v && a.v) );
    for(int i=0;i<n;i++)
       v[i]=a.v[i];
}

template <class T>
void Vect<T>::Affiche(char * Message) const
{   cout<<Message<<endl;
    for(int i=0;i<n;i++)
    cout<<"Element i="<<i<<"\t"<<v[i]<<endl;
}

template <class U>
void InitVect(Vect<U> & v)
{  for(int i=0;i<v.n;i++)
     v[i]= (U)i/v.n;
}

typedef Vect<double> VectR;

void main()
{   Vect<double> w(3);
    VectR        u(3);
    InitVect(w);   w.Affiche("---- vecteur w ----");
                   u.Affiche("---- vecteur u ----");
    u=w; u[0]=10;  u.Affiche("---- vecteur u ----");
    Vect<int>    s(3);
    InitVect(s);   s.Affiche("---- vecteur s ----");
}
```

Le tableau est entièrement défini par sa dimension `n` et un pointeur de type `T`. Le rôle du constructeur sera alors d'allouer `n` locations mémoire du type `T`. C'est le cas du premier constructeur qui prend comme argument un entier et qui initialise le pointeur `v` en utilisant l'opérateur `new`, comme nous l'avons fait pour les tableaux dynamiques. L'initialisation des données membres étant faite par la liste d'initialisation, le corps du constructeur aurait pu rester vide. Il contient néanmoins une instruction de type `assert` (assertion) qui nous assure que la valeur de `n` est non-nulle et que `v` a été alloué. Les assertions[4] ont la forme générale `assert(cond)` et provoque l'arrêt de l'exécution du programme si la condition `cond` n'est pas satisfaite.

Remarque 1.16 *Il est vivement recommandé d'utiliser souvent les assertions pour prévenir les erreurs d'exécution et pour aider au débogage des programmes.*

[4]Dont le prototype est inclus par la directive de préprocesseur `#include <cassert>`.

Le deuxième constructeur a comme argument une référence (constante) d'un objet de type `Vect`, donc va créer un nouvel objet par copie. L'espace mémoire nécessaire est alloué par la liste d'initialisation, suivant la dimension extraite de `w` par la syntaxe `w.n` (on utilise l'opérateur « `.` », car il s'agit d'une référence). Une fois la mémoire allouée, il reste à copier effectivement les valeurs de `w` dans l'objet courant. Cette opération est effectuée d'un seul coup en utilisant le pointeur `this` de la classe ; comme il s'agit de modifier sa valeur, c'est `(*this)` qui va prendre la valeur de `w`. Mais, attention, pour réaliser cette attribution, il faut définir l'opérateur « `=` » qui sera recherché par le compilateur[5]. Cet opérateur, défini plus loin, va faire la copie élément par élément de l'objet indiqué en argument. Rappelons (remarque 1.13) que l'opérateur « `=` » manipule des objets parfaitement définis, et il n'a pas besoin d'allouer de la mémoire comme le constructeur par copie.

En même temps, l'opérateur « `=` » fait appel à l'opérateur crochet « `[]` » qui doit être défini à son tour. Son rôle est de renvoyer la référence d'un élément de tableau à partir de son indice. La définition de cet opérateur constitue l'endroit idéal pour imposer une protection contre les dépassements de tableaux (voir l'assertion correspondante).

Avec ces deux opérateurs, nous pouvons accéder (voir `main`) et modifier un élément de tableau (`u[0]=10;`), ou même effectuer des attributions *en vectoriel* (`u=w;`). Le programme va fournir les résultats suivants :

```
---- vecteur w ----                              //    déclaré de type double
Element i=0 0                                   //    initialisé par InitVect
Element i=1 0.333333
Element i=2 0.666667
---- vecteur u ----                         //    juste déclaré de type double
Element i=0 0
Element i=1 0
Element i=2 0
---- vecteur u ----                              //    résultat de u=w;u[0]=10
Element i=0 10
Element i=1 0.333333
Element i=2 0.666667
---- vecteur s ----                                 //    déclaré de type int
Element i=0 0                                   //    initialisé par InitVect
Element i=1 0
Element i=2 0
```

[5] Voir aussi les règles de programmation, §**??**.