Université Paris-Dauphine

Cours C++ pour programmeurs Java

M1

Tristan Cazenave

2010

Résumé

Programmation Objet en C++.

Table des matières

Chapitre 1

Opérations de base en C++

1.1 Bibliographie

Programmation Objet en langage C++, Alexandre Guidet, Ellipses.

Exercices en langage C++, Claude Delannoy, Eyrolles.

1.2 Un programme minimal

C++ est un langage qui ajoute des fonctionnalités objet au langage C. C'est donc un sur ensemble d'un langage impératif qui n'a pas été conçu dès le début pour être orienté objet. Ainsi en C++ on peut définir un programme en dehors de toute classe contrairement à Java.

Un programme minimal peut s'écrire :

```
#include <iostream>

int main (int argc, char *argv[])  {
  std::cout << "bonjour" << std::endl;

  return 0;
}
```

Ce programme affiche bonjour sur la sortie standard.

La première ligne sert à utiliser la bibliothèque d'entrées/sorties. C'est une instruction du préprocesseur pour inclure l'en-tête des fonctions de la librairie d'entrees/sorties. Les entrées/sorties sont considérées comme des flots.

Un programme peut être appelé avec des arguments, par exemple :

```
#include <iostream>
```

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

  if (argc > 0)
    std::cout << argv [1] << std::endl;
  else
    std::cout << "donnez un argument" << std::endl;

  return 0;
```

Dans ce programme char *argv [] représente un tableau de chaînes de caractères. Une chaîne de caractères en C est un tableau de caractères, un tableau de chaînes de caractères est donc un tableau de tableaux de caractères.

Pour représenter un tableau on peut écrire indifféremment :

```
char tableau [100];
char tableau[];
char *tableau;
```

Dans le premier cas le tableau est alloué et occupe 100 octets, dans les deuxième et troisième cas le tableau n'est pas alloué, seul la place pour un pointeur est allouée.

Pour ne pas avoir à écrire std : :cout mais simplement cout pour le flot de sortie, on peut utiliser l'instruction using namespace std, le programme devient alors :

```
#include <iostream>

using namespace std;

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

  if (argc > 0)
    cout << argv [1] << endl;
  else
    cout << "donnez un argument" << endl;

  return 0;
```

1.3 Le préprocesseur

Le préprocesseur permet d'exécuter des instructions avant la compilation du programme.

L'instruction la plus courante est *#include*, elle peut être utilisée pour inclure une bibliothèque standard comme pour l'instruction *#include <iostream>*, mais aussi pour inclure vos propres fichiers d'en-tête avec par exemple *#include "MaClasse.h"*.

Une macro qui définit ou remplace une expression est *#define*.

La définition d'une expression peut être utile lorsqu'un fichier en-tête est inclus dans plusieurs fichiers en-têtes. On ne veut alors l'inclure qu'une seule fois. Pour cela on utilise les instructions suivantes du preprocesseur dans le fichier en-tête :

```
#ifndef MACLASSE_H
#define MACLASSE_H
void maFonction ();
#endif
```

La première fois que le fichier est inclus il définira maFonction (), les fois suivantes il ne prendra pas cette définition.

On peut aussi associer une constante avec un *#define* :

```
#define MAX_SIZE 65536
#define PI 3.141162654
```

Toutefois en C++, on préférera le mot-clé const pour définir une constante :

```
const int MaxSize = 65536;
```

Le préprocesseur permet aussi de donner des directives de compilation :

```
#if DEBUGLEVEL = 1
// code pour debugger au niveau 1
#elseif DEBUGLEVEL = 2
// code pour debugger au niveau 2
#endif
```

On peut aussi remplacer des expressions à l'aide du préprocesseur :

```
#define maxi(a,b) ((a)>(b)?(a):(b))
#define abs(a) ((a)>0?(a):-(a))
```

1.4 Les opérateurs new et delete

En C++ on alloue dynamiquement de la mémoire avec l'opérateur new. Par exemple pour créer un tableau de 20 entiers on écrit :

```
int * tableau = new int [20];
```

Pour allouer un seul entier on écrira :

```
int * entier = new int;
```

Contrairement à Java qui possède un garbage collector, on doit explicitement libérer la mémoire en C++. Ceci se fait à l'aide de l'opérateur delete.

Pour libérer l'espace d'un tableau, on fait :

```
delete [] tableau;
```

Pour libérer un seul entier, on fait :

```
delete entier;
```

1.5 Les fonctions

Une fonction peut être déclarée sans lien avec une classe. Elle peut être définie dans un autre fichier du moment que son en-tête est donné dans le fichier qui l'utilise.

Par exemple la fonction suivante :

```
int mini (int a, int b) {
  if (a < b)
    return a;
  else
    return b;
}
```

peut être définie dans le fichier mini.c. La fonction mini peut être utilisée dans un autre fichier du moment qu'on la déclare avant de l'utiliser avec :

```
int mini (int a, int b);
```

Le compilateur retrouvera son emplacement dans le programme à l'édition des liens.

Le passage de paramètres à une fonction se fait par défaut par valeur, ce qui veut dire que le paramètre est recopié dans une nouvelle variable lorsqu'il est passé. Pour que le paramètre soit modifié au retour de la fonction il faut le passe par référence :

```
#include <iostream>

using namespace std;

void plusUn (int a) {
  a++;
```

```
}

void plusUnReference (int & a) {
  a++;
}

int main () {
  int b = 0;
  plusUn (b);
  cout << b << endl; // affiche 0
  plusUnReference (b);
  cout << b << endl; // affiche 1
}
```

Lors du passage par valeur, le paramètre est recopié dans une nouvelle variable. Lorsqu'on passe des objets volumineux on peut vouloir ne pas recopier le paramètre. On utilise alors un passage par référence constant : l'objet n'est pas recopié lors du passage mais on s'assure avec le mot-clé const qu'il n'est pas modifié dans la fonction. Exemple :

```
void plusB (int & a, const int & b) {
  a += b;
}
```

On peut aussi passer un pointeur sur un objet à une fonction, de cette façon l'objet n'est pas recopié mais seulement le pointeur. On a alors la fonction suivante :

```
void plusB (int * a, int * b) {
  *a += *b;
}
```

Lors de l'appel de la fonction il faut donner l'adresse des variables :

```
int a, b;
plusB (&a, &b);
```

Lorsqu'on définit une fonction, on peut donner des valeurs par défauts aux paramètres et les rendre optionnels lors de l'appel de la fonction. Si on veut par exemple que la fonction plusB ajoute 1, on écrira :

```
void plusB (int & a, int b = 1) {
  a += b;
}
```

1.6 break, continue et goto

break est une instruction qui permet de sortir d'une structure de contrôle. Elle est utilisée pour sortir d'une boucle ou pour sortir d'un switch.

Par exemple si on veut sortir d'un boucle infinie lorsqu'on lit le nombre 100 au clavier, on peut écrire :

```
while (true) {
  int n;
  cin >> n;
  if (n == 100)
    break;
}
```

De même si on est dans une liste de choix gérés par un switch, il faut utiliser break pour sortir de la suite d'instructions :

```
switch (n) {
  case 1 :
    cout << "un";
    break;
  case 2 :
    cout << "deux";
    break;
  default :
    cout << "beaucoup";
}
```

Lorsqu'on est dans une boucle et qu'on veut sauter des éléments, on peut utiliser l'instruction *continue*. Par exemple si on veut afficher tous les entiers de 1 à 100 sauf 50, on peut écrire :

```
for (int i = 1; i <= 100; i++) {
  if (i == 50)
    continue;
  cout << i << endl;
}
```

Lorsqu'on veut aller directement à un autre endroit du code, on peut utiliser *goto* :

```
if (i == 1) {
  cout << "un";
  goto fin;
}
if (i == 2) {
  cout << "deux";
  goto fin;
}
cout << "beaucoup";
fin: // une etiquette
```

1.7 Crible d'Eratosthène

Comme premier exemple de programme on va écrire un programme qui trouve tous les nombres premiers inférieurs à un nombre donné. Le principe est le suivant, quand on trouve un nouveau nombre premier on élimine tous ses multiples des nombres à tester. De cette façon le prochain nombre à tester n'a pas de diviseur et il est donc premier.

```
#include <iostream>

using namespace std;

void erat (int maxi) {
  bool * possible;

  possible = new bool [maxi];

  for (int i = 0; i < maxi; i++)
    possible [i] = true;

  for (int i = 2; i < maxi; i++)
    if (possible [i]) {
      cout << i << " ";
      int n = i;
      while (n < maxi) {
        possible [n] = false;
        n += i;
      }
    }
  cout << endl;

  delete [] possible;
}

int main (int argc, char **argv) {
  erat (1000000);
}
```

1.8 Les chaînes de caractères

La chaines de caractères ont leur classe spécifique en C++, toutefois de nombreuses méthodes existent en C qui traitent les chaines de caractères comme des tableaux.

1.8.1 Les chaînes de caractères en C

Les chaînes de caractères en C sont traitées comme des tableaux de caractères. Les fonctions pour les chaînes de caractères C sont disponibles dans string.h. Le

caractère '\0' est le caractère de fin de chaîne, et '\n' est le retour à la ligne.

On peut déclarer les chaines de caractères comme un pointeur sur un caractère :

```
const char * mot = "analphabete";
```

On peut aussi initialiser un tableau de caractères avec une chaine :

```
char mot [30] = "analphabete";
```

On peut laisser au compilateur le soin de dimensionner le tableau :

```
char mot [] = "analphabete";
```

Plutot que de donner une chaine entre guillemets, il est équivalent mais moins pratique de donner directement le tableau :

```
char mot [30] = {'a', 'n', 'a', 'l', 'p', 'h', 'a', 'b', 'e', 't', 'e', '\0'};
```

Les fonctions de string.h

*size_t strlen(const char *) ;* renvoie la taille de la chaîne de caractères passée en paramètre. Par exemple strlen ("toto") renvoie 4, et strlen (mot) renvoie 11.

*int strcmp(const char *, const char *) ;* compare deux chaînes dans l'ordre alphabétique. Si la première chaine est plus petite que la deuxième elle renvoie une valeur négative, si elle est plus grande elle renvoie une valeur positive. Si les deux chaînes sont égales elle renvoie zéro. Par exemple *strcmp ("tata", "toto")* est négatif alors que *strcmp ("toto", "toto")* renvoie zéro.

La fonction *char *strcat(char *dest, const char *src) ;* ajoute la chaîne src à la fin de la chaîne dest.

```
#include <iostream>
#include <string.h>

using namespace std;

int main(int argc, char **argv) {
  char s [100] = "bonjour";

  if (argc > 1) {
    cout << "avant strcat : " << s << endl;
    strcat(s ,argv [1]);
    cout << "apres strcat : " << s << endl;
  }
}
```

La fonction *char *strncat(char *dest, const char *src, size_t n) ;* ajoute au plus n caractères de la chaine src.

D'autres fonctions recherchent un sous chaine dans une chaine. Ces fonctions renvoient NULL si la sous-chaine n'est pas présente dans la chaine, sinon elles renvoient un pointeur sur le début de la sous-chaine.

La fonction *char *strstr(const char *botteDeFoin, const char *aiguille) ;* recherche la sous-chaine aiguille dans la chaine botteDeFoin.

La fonction *char *strchr(const char *s, int c) ;* recherche le caractère c dans la chaine s.

La fonction *char *strcpy(char *dest, const char *src) ;* copie la chaine src vers dest.

Enfin il existe des fonctions de copie et d'initialisation de chaînes. Ces fonctions sont souvent utilisées en dehors de leur application aux chaines pour copier des zones mémoire.

La fonction *void *memcpy(void *dest, const void *src, size_t n) ;* copie n octets de src vers dest.

La fonction *void *memset(void *dest, int c, size_t n) ;* copie n fois de suite l'octet c après l'adresse dest.

```
#include <iostream>
#include <string.h>

using namespace std;

int main () {
  const char *s = "Bonjour!\n";
  cout << s;
  char chaine [500];
  // initialiser la chaine chaine avec des 0
  memset (chaine, 0, 500);
  // copier la chaine s dans la chaine chaine
  // strlen (s) etant la taille de s
  // et le nombre de caractères à recopier
  memcpy (chaine, s, strlen (s));
  // ajouter s a la fin de chaine
  strcat (chaine, s);
  if (!strcmp (chaine, "Bonjour!\nBonjour!\n"))
    cout << "ok\n";
  if (!strncmp (chaine, "Bonjour!\nHello!\n", 5))
    cout << "ok\n";
  strcpy (chaine, s);
  cout << chaine;
}

// le resultat est :
// Bonjour!
// ok
```

```
// ok
// Bonjour!
```

Les fonctions de stdio.h

La bibliothèque d'entrée sorties de C est *stdio.h*. Elle comprend aussi des fonctions de traitement des chaînes de caractères. Une fonction très utile est *sprintf (char * destination, char *format, arguments)*. Par exemple pour insérer des entiers, de float ou des caractères dans une chaîne on peut utiliser :

Une autre fonction très utile est *sscanf (char * source, char *format, arguments)* qui permet de récupérer des valeurs des types fondamentaux depuis une chaine.

```
#include <iostream>
#include <stdio.h>

using namespace std;

int main () {
  char s [1000];
  sprintf (s, "J'affiche %d et %1.9f\n", 36, 3.141592654);
  cout << s << endl;
  int i;
  float f;
  sscanf (s, "J'affiche %d et %1.9f\n", &i, &f);
  cout << i << " " << f << endl;
}
```

Les fonctions de stdlib.h

La bibliothèque C *stdlib.h* contient elle aussi des fonctions de conversion. Par exemple *int atoi(const char *str)* qui convertit un chaîne vers un entier ou *double atof(const char *str)* qui convertit une chaîne vers un double.

D'autres fonctions utiles de *stdlib.h* sont par exemple *int rand()* qui génère un nombre aléatoire entre 0 et RAND_MAX, et *void srand(unsigned int seed)* qui initialise le générateur de nombres aléatoires.

Le pendu

Pour illustrer l'utilisation des chaines de caractères C, on peut écrire un programme qui joue au pendu :

```
#include <iostream>
#include <string.h>
#include <stdlib.h>

using namespace std;
```

```
bool estPresente (char lettre, const char *mot) {
  for (int i = 0; i < strlen (mot); i++)
    if (mot [i] == lettre)
      return true;
  return false;
}

int main (int argc, char **argv) {
  const char * mot = "analphabete";

  if (argc > 1) {
    mot = argv [1];
  }

  char *motif = new char [strlen (mot) + 1];
  for (int i = 0; i < strlen (mot); i++)
    motif [i] = '*';
  motif [strlen (mot)] = '\0';

  int essais = 5, trouves = 0;
  while (true) {
    cout << "motif du mot = " << motif << endl;
    cout << "vous avez " << essais << " essais" << endl;
    cout << "donnez une lettre ";
    char lettre;
    cin >> lettre;
    if (estPresente (lettre, mot)) {
      for (int i = 0; i < strlen (mot); i++)
        if ((mot [i] == lettre) && (motif [i] == '*')) {
          motif [i] = lettre;
          trouves++;
        }
    }
    else
      essais--;
    if (trouves == strlen (mot)) {
      cout << "bravo !" << endl;
      break;
    }
    if (essais == 0) {
      cout << "perdu !" << endl;
      break;
    }
  }
}
```

1.8.2 Les chaînes de caractères en C++

En C++, le type string permet de gérer les chaînes de caractères. Il permet d'utiliser des fonctions de plus haut niveau que les chaînes de caractères C, et de les écrire simplement. De plus on peut toujours revenir aux chaînes à la C. Pour

utiliser les string C++ il faut inclure l'en tête *string*.

Les strings C++ sont plus pratiques et plus sures que les chaines de caractères C car elle gère la mémoire à l'intérieur de la classe string.

On peut toujours revenir aux chaines C en utilisant la méthode *char * c_ str ()*.

On peut additionner les strings avec des strings ou des chaines de caractères.

On peut comparer les strings avec ==, > et <.

On peut affecter des chaines de caractères à des strings.

On peut récupérer un caractère avec l'opérateur []. Par exemple pour une string s = "bonjour", on peut récupérer s [3] qui vaudra 'j'.

Enfin on peut appliquer les opérateurs d'entrée/sortie aux strings et les rediriger vers cout ou cerr, et les lire avec cin.

Exemple de programme :

```
#include <iostream>
#include <string>

using namespace std;

int main () {
  string s = "bonjour";
  cout << "Quel est votre prenom : ";
  string prenom;
  cin >> prenom;
  if (prenom == "tristan")
    s = s + " maitre";
  else
    s = s + " " + prenom;
  cout << s << endl;
}
```

Pour comparer une string à une autre on utilise *int compare (const string& str) const ;*. La fonction retourne 0 si les chaines sont égales, une valeur négative si l'objet appelant est plus petit alphabétiquement, une valeur positive dans le cas contraire.

On peut également comparer une sous chaine avec *int compare (size_t pos1, size_t n1, const string& str) const ;* en commençant à pos1 sur une longueur de n1.

On peut aussi comparer avec des chaines de caractères C.

Exemple d'utilisation :

```
// comparing apples with apples
#include <iostream>
```

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

int main ()
{
  string str1 ("green apple");
  string str2 ("red apple");

  if (str1.compare(str2) != 0)
    cout << str1 << " is not " << str2 << "\n";

  if (str1.compare(6,5,"apple") == 0)
    cout << "still, " << str1 << " is an apple\n";

  if (str2.compare(str2.size()-5,5,"apple") == 0)
    cout << "and " << str2 << " is also an apple\n";

  if (str1.compare(6,5,str2,4,5) == 0)
    cout << "therefore, both are apples\n";

  return 0;
}
```

```
green apple is not red apple
still, green apple is an apple
and red apple is also an apple
therefore, both are apples
```

Pour insérer une string dans une autre on utilise *string& insert (size_t pos1, const string& str);*. La fonction insère la string str à partir de la position pos1.

On peut aussi n'insérer qu'une partie de la chaine passée en paramètre avec *string& insert (size_t pos1, const string& str, size_t pos2, size_t n);* qui insère à partir de pos2 sur n caractères.

Enfin on peut insérer des chaines C et on peut repérer les indices dans une chaine avec des itérateurs.

Exemple d'utilisation :

```
// inserting into a string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str="to be question";
  string str2="the ";
  string str3="or not to be";
  string::iterator it;

  // used in the same order as described above:
```

```
  str.insert(6,str2);                 // to be (the )question
  str.insert(6,str3,3,4);             // to be (not )the question
  str.insert(10,"that is cool",8);    // to be not (that is )the question
  str.insert(10,"to be ");            // to be not (to be )that is the question
  str.insert(15,1,':');               // to be not to be(:) that is the question
  it = str.insert(str.begin()+5,','); // to be(,) not to be: that is the question
  str.insert (str.end(),3,'.');       // to be, not to be: that is the question(...)
  str.insert (it+2,str3.begin(),str3.begin()+3); // (or )

  cout << str << endl;
  return 0;
}

to be, or not to be: that is the question...
```

On peut ajouter une string, une chaine C ou un caractère à une string avec l'opérateur +=. Par exemple :

```
String s = "toto";
s += " va a la plage\n";
cout << s;
```

Pour extraire une sous chaine d'une chaine de caractères, on utilise *size_t copy (char* s, size_t n, size_t pos = 0)* qui va remplir la chaine s avec les caractères compris entre pos et pos+n de la string appelante.

Voici une exemple d'utilisation des fonctions que nous avons vues :

```
#include <iostream>
#include <string>
#include <stdlib.h>

using namespace std;

int main () {
  string s;
  cout << "Entrez une chaine : ";
  cin >> s;
  char chaine [100];
  sprintf (chaine, "%d", (int)s.size ());
  string s1 = "La chaine " + s + " a " + chaine + " caracteres";
  cout << s1 << endl;
  // on insere Bonjour au debut
  s1.insert (0, "Bonjour, ");
  cout << s1 << endl;
  s1.copy (chaine, 10);
  chaine [10] = '\0';
  cout << chaine << endl;
}

// Entrez une chaine : toto
```

```
// La chaine toto a 4 caracteres
// Bonjour, La chaine toto a 4 caracteres
// Bonjour, L
```

1.9 Les entrées/sorties

1.9.1 Les entrées sorties en C

En C, les entrées sorties peuvent être traitées comme des fichiers. La structure C qui représente un fichier est FILE.

Pour lire des informations on utilise *fscanf (FILE *fp, const char * format, arguments)*.

Le clavier correspond à stdin et au descripteur de fichier numéro 0. Pour lire un entier au clavier on utilisera :

```
int entier;
fscanf (stdin, "%d", &entier);
```

Remarquez qu'il faut passer l'adresse de l'entier à remplir pour que sa valeur soit modifiée au retour de la fonction fscanf.

Pour écrire des informations on utilise *fprintf (FILE *fp, const char * format, arguments)*.

L'écran correspond à stdout et au descripteur de fichier numéro 1. La sortie d'erreur est aussi l'écran par défaut, elle correspond à stderr et au descripteur de fichier numéro 2. Pour écrire un double à l'écran on utilisera :

```
double d;
fprintf (stdout, "%f", d);
```

Remarquez que pour fprintf le passage se fait par valeur car la paramètre n'a pas besoin d'être modifié par la fonction fprintf.

Il existe une fonction qui fait automatiquement des sorties à l'écran : *printf (const char * format, arguments)*. Et une fonction qui lit automatiquement au clavier : *scanf (const char * format, arguments)*.

1.9.2 Les entrées sorties en C++

Les entrées et les sorties sont considérées comme des flots. Un flot est associé à un fichier ou à un périphérique.

Les flots d'entrée sont de type *istream*, comme par exemple *cin*. Pour récupérer les informations depuis un flot d'entrée on utilise l'opérateur ». L'opérateur » renvoie le flot, on peut donc l'utiliser en cascade.

Le clavier est un istream qui se nomme *std : :cin*.

Les flots de sortie sont de type *ostream*, comme par exemple *cout*. Pour récupérer les informations depuis un flot d'entrée on utilise l'opérateur «, de même que » il peut être utilisé en cascade.

L'écran est un ostream qui se nomme *std : :cout*.

La sortie d'erreur est un ostream qui se nomme *std : :cerr*.

La classe iostream permet de combiner flots de sortie et flots d'entrée.

On peut donner des instructions de formatage à un flot. On a les méthodes suivantes :

- setw (int) qui définit le nombre de caractères sur lequel va se faire l'affichage.
- setprecision (int) qui donne la précision des nombres flottants.
- setbase (int) qui définit la base dans laquelle afficher le nombre.

Le programme suivant affiche un réél entré au clavier sur 10 caractères avec une précision variable :

```
#include <iostream>
#include <iomanip>

using namespace std;

int main () {
  double d;
  cout << "Entrez un reel : ";
  cin >> d;
  for (int i = 0; i < 10; i++)
    cout << setw (10) << setprecision (i) << d << endl;
}
```

1.10 Les fichiers

1.10.1 Les fichiers en C avec des fonctions de haut niveau

Pour ouvrir un fichier en lecture ou en écriture on utilise *FILE * fopen (char * name, char * mode)*. Le paramètre format vaut "r" pour une ouverture en lecture, "w" pour une ouverture en écriture, "a" pour une ouverture en ajout.

fopen renvoie NULL si le fichier ne peut pas être ouvert.

Pour écrire les cent premiers entiers dans un fichier toto on écrit :

```
include <stdio.h>

int main() {
```

```
  const char * nom = "toto";
  FILE * fp = fopen(nom, "w");
  if (fp != NULL) {
    for (int i = 0; i < 100; i++)
      fprintf (fp, "%d ", i);
    fprintf (fp, "\n");
    fclose (fp);
  }
}
```

Pour lire les cent premiers entiers depuis le fichier toto on écrit :

```
include <stdio.h>

int main() {
  int entiers [100]
  const char * nom = "toto";
  FILE * fp = fopen(nom, "r");
  if (fp != NULL) {
    for (int i = 0; i < 100; i++)
      fscanf (fp, "%d ", &entier [i]);
    fclose (fp);
  }
}
```

1.10.2 Les fichiers en C avec des fonctions de bas niveau

Les opérations de bas niveau pour accéder aux fichiers sont *open (), close (), read () et write ()*.

Pour utiliser ces opérations on doit inclure *<sys/types.h>*, *<sys/stat.h>*, *<unistd.h>*, *<fcntl.h>*.

La fonction pour ouvrir un fichier est *int open (const char * nom, int options) ;*. Elle retourne -1 en cas d'erreur et le descripteur du fichier en cas de succés.

Les options sont des bits qu'on peut mettre à un ou zéro suivant le mode d'ouverture du fichier. Ces bits correspondent à des acronymes qui peuvent être combinés avec des ou logique (|). Les acronymes sont :

- O_RDONLY pour ouvrir en lecture,
- O_WRONLY pour ouvrir en écriture
- O_RDWR pour ouvrir en lecture écriture
- O_CREAT pour créer le fichier s'il n'existe pas,
- O_EXCL pour faire échouer l'appel si le fichier existe déjà,
- O_TRUNC pour réinitialiser un fichier qui existe déjà,
- O_APPEND pour ajouter à la fin du fichier.

La fonction pour fermer un fichier est *int close (int fd) ;*. Elle ferme le fichier qui a pour descripteur fd.

La fonction pour lire un fichier est *int read (int fd, void * buffer, size_ t nombre) ;*.

Elle permet de lire le fichier qui a pour descripteur fd. Elle lit au plus nombre octets qu'elle place à partir de l'adresse buffer. Elle renvoie -1 en cas d'erreur et le nombre d'octets lu en cas de succès.

La fonction pour écrire dans un fichier est *int write (int fd, const void * buffer, size_ t nombre) ;*. Elle renvoie -1 en cas d'erreur et le nombre d'octets écrits en cas de succès. Elle écrit au plus nombre octets dans le fichier de descripteur fd en partant de l'adresse buffer.

On peut maintenant écrire un programme qui effectue la copie d'un fichier à l'aide de ces instructions :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char * argv []) {
  int fd1, fd2;
  char buffer [10000];
  int nb;

  if (argc != 3) {
    fprintf (stderr, "%s fichier1 fichier2\n", argv [0]);
    exit(1);
  }

  fd1 = open (argv[1], O_RDONLY);
  if (fd1 == -1) {
    fprintf (stderr, "erreur lecture %s\n", argv [1]);
    exit(1);
  }

  fd2 = open (argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd2 == -1) {
    fprintf (stderr, "erreur ecriture %s\n", argv [2]);
    exit(1);
  }

  nb = read (fd1, buffer, 10000);
  while (nb > 0) {
    write (fd2, buffer, nb);
    nb = read (fd1, buffer, 10000);
  }

  close(fd1);
  close(fd2);

  return 0;
}
```

On peut aussi se déplacer à l'intérieur d'un fichier avec la fonction *int lseek (int fd, int deplacement, int mode) ;*. Le déplacement souhaité en nombre d'octets est dans la variable deplacement. La fonction renvoie le déplacement réellement effectué. Les trois modes d'appel de la fonction sont possible :

- SEEK_SET pour commencer au début du fichier ;
- SEEK_CUR pour commencer à la position courante ;
- SEEK_END pour commencer à la fin du fichier.

1.10.3 Les fichiers en C++

Les fichiers sont des flots particuliers.

Pour accéder aux fichiers il faut inclure l'en tête *#include <fstream>*.

Les écritures sur un fichier se font avec un objet *ofstream*. Pour ouvrir un fichier on appelle le constructeur *ofstream (const char * filename, mode)*.

Les différents modes d'ouverture sont :

- ios : :app (append) ajout à la fin du fichier.
- ios : :ate (at end) met l'index à la fin du fichier.
- ios : :binary (binary) Pour ouvrir un fichier binaire.
- ios : :in (input) permet la lecture
- ios : :out (output) permet l'écriture.
- ios : :trunc (truncate) vide le fichier à l'ouverture.

Pour écrire des entrées sorties formatées dans un fichier, on utilise l'opérateur «.

Pour lire des entrées sorties formatées, on utilise ».

Pour fermer un fichier on utilise la méthode close ().

Pour écrire n'importe quelles données, on les convertit vers un tableau de caractères et on utilise la méthode *ostream& write (const char* s , streamsize n)*.

Pour lire des données d'un fichier on utilise la méthode *istream& read (char* s, streamsize n)*.

Pour se positionner à l'intérieur d'un fichier on utilise *istream& seekg (streamoff distance, ios_base : :seekdir dir)* où distance représente le nombre de caractères dont on se déplace, la direction et l'endroit de départ sont spécifiés par dir.

- Si dir vaut *ios_base : :beg* on part du début du fichier.
- Si dir vaut *ios_base : :cur* on part de la position courante.
- Si dir vaut *ios_base : :end* on part de la fin du fichier.

Pour récupérer l'indice de la position dans un fichier on utilise *streampos tellg ()*

Par exemple pour copier un fichier on utilise le programme suivant :

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

int main (int argc, char *argv []) {
  char * buffer;
  long size;

  if (argc == 3) {
    ifstream entree (argv [1], ifstream::binary);
    ofstream sortie (argv [2], ofstream::binary);

    // trouver la taille du fichier
    entree.seekg (0, ifstream::end);
    size=entree.tellg ();
    entree.seekg (0);

    // allocation de la mémoire
    buffer = new char [size];

    // lecture du fichier
    entree.read (buffer, size);

    // ecriture du fichier
    sortie.write (buffer, size);

    // liberation de la memoire
    delete [] buffer;

    entree.close ();
    sortie.close ();
  }
  return 0;
}
```

Une fonction qui permet de lire un fichier texte ligne par ligne est la fonction *bool getline (ifstream &, string & line)*.

On peut alors écrire un programme qui lit un fichier texte et le met en majuscules :

```
#include <fstream>
#include <ctype.h>

using namespace std;

int main (int argc, char *argv []) {
  string s;
  if (argc == 3) {
    ifstream entree (argv [1]);
    ofstream sortie (argv [2]);
    while (getline (entree, s)) {
      for (int i = 0; i < s.size (); i++)
        s [i] = toupper (s [i]);
```

```
      sortie << s << endl
    }
    entree.close ();
    sortie.close ();
  }
  return 0;
}
```

Chapitre 2

TD 1

2.1 Devinette

Ecrire un programme qui fait deviner un nombre entre 0 et 20.

2.2 Maximum

Ecrire deux fonctions qui trouvent le maximum d'un tableau, les arguments seront passés dans un cas par reference et dans l'autre cas par adresse.

2.3 Monte-Carlo PI

Ecrire un programme qui calcule une approximation de π avec la méthode de Monte-Carlo. Pour cela on choisit au hasard un point dans un carré de coté 2 centré sur l'origine et on regarde s'il est dans le cercle de rayon 1 centré sur l'origine. La surface du disque de rayon 1 étant π, la probabilité qu'un point tiré au hasard dans le carré soit dans le cercle est égale à la surface du disque divisée par la surface du carré, soit $\frac{\pi}{4}$. Vous afficherez le résultat avec une précision de 8 décimales.

2.4 La conjecture de Goldbach

La conjecture de Goldbach suppose que tout nombre pair strictement supérieur à 2 peut s'écrire comme la somme de deux nombre premiers. Ecrire un programme qui vérifie si la conjecture de Goldbach est vrai pour les nombres pairs inférieurs à un nombre donné.

2.5 itoa

Ecrire un fonction qui renverse une chaine de caractères C. Ecrire ensuite une fonction *void itoa(int n, char *s)* qui convertit un entier vers une chaine de caractères C.

2.6 Conjugaison

Ecrire un programme qui conjugue un verbe du premier groupe entré au clavier en utilisant des string.

2.7 Le compte est bon

Ecrire un programme qui joue au compte est bon avec au plus 100 entiers. Le but est de trouver une expression qui vaut un nombre prédéfini. L'expression ne peut contenir que les entiers passés en argument et les quatres opérations de base : +, -, *, /.

On commencera par ecrire la fonction main qui récupère les données du problème avec les arguments du programme.

On ecrira ensuite une fonction qui affiche sous forme agréable une chaine de caractères qui représente une expression parcourue en racine-gauche-droite.

On ecrira ensuite une fonction qui évalue cette chaine de caractères.

On écrira enfin la fonction qui engendre et teste les expressions.

2.8 Opérations sur un fichier

Ecrire un programme qui demande un nombre au clavier et qui écrit un fichier contenant les entiers entre 0 et ce nombre.

Ecrire un programme qui lit ce fichier et qui affiche tous ses entiers.

Ecrire un programme qui affiche l'entier du fichier dont l'indice est donné au clavier.

Refaire les mêmes opérations pour un fichier binaire.

Chapitre 3

La programmation objet

3.1 Structures de données en C

En C pour regrouper des informations dans une même structure on utilise le mot-clé struct.

Par exemple pour représenter un rationnel on peut déclarer :

```
struct Rationnel {
  int _n, _d;
};
```

Pour déclarer un rationnel on utilise alors :

```
struct Rationnel r;
```

On peut alors écrire des fonctions sur les rationnels en passant la structure par valeur :

```
void affiche (FILE *fp, struct Rationnel r) {
  fprintf (fp, "%d / %d", r._n, r._d);
}
```

On peut écrire la même fonction en passant la structure par adresse :

```
void affiche (FILE *fp, struct Rationnel * r) {
  fprintf (fp, "%d / %d", r->_n, r->_d);
}
```

En C++ on utilisera plutôt des classes qui fournissent plus de fonctionnalités que les structures C.

3.2 Les classes

3.2.1 Une classe simple

La manière la plus simple d'écrire une classe est d'écrire directement les fonctions dans la déclaration de la classe. Les données et les méthodes de la classe peuvent être privées ou publiques. Lorsqu'on déclare des éléments d'une classe, on peut les affecter à d'autre éléments et C++ recopiera la zone mémoire de l'élément à recopier. Par exemple si on veut écrire une classe rationnel :

```
#include <iostream>

using namespace std;

class Rationnel {
 private :
  int _n, _d;

 public :
  Rationnel (int n, int d) {
    _n = n;
    _d = d;
    simplification (pgcd ());
  }

  void simplification (int p) {
    _n /= p;
    _d /= p;
  }

  int pgcd () {
    int a = _n, b = _d;
    int reste, tampon;
    if (a < b){
      tampon = a;
      a = b;
      b = tampon;
    }
    while (b != 0){
      reste = a % b;
      a = b;
      b = reste;
    }
    return a;
  }

  void inverser () {
    int tmp = _n;
    _n = _d;
    _d = tmp;
  }
```

```
  void affiche () {
    cout << _n << "/" << _d << endl;
  }
};

int main () {
  Rationnel a (1, 2);
  Rationnel b = a;
  b.inverser ();
  b.affiche ();

  Rationnel c = new Rationnel (5, 25);
  c->inverser ();
  c->affiche ();
  b = *c;
}
```

3.2.2 Contenu des méthodes à l'extérieur de la classe

Il est courant de réutiliser une classe dans d'autres fichiers d'un projet. Dans ce cas on sépare la déclaration de la classe du contenu des méthodes. Par exemple pour la classe Rationnel on aura le fichier en tête "Rationnel.h" suivant :

```
#ifndef RATIONNEL_H
#define RATIONNEL_H

class Rationnel {
 private :
  int _n, _d;

 public :
  Rationnel (int n, int d);
  void simplification (int p);
  int pgcd ();
  void inverser ();
  void affiche ();
};

#endif
```

On a le contenu des méthodes dans le fichier "Rationnel.c" suivant :

```
#include <iostream>

#include "Rationnel.h"

using namespace std;

Rationnel::Rationnel (int n, int d) {
  _n = n;
```

```
  _d = d;
  simplification (pgcd ());
}

void Rationnel::simplification (int p) {
  _n /= p;
  _d /= p;
}

int Rationnel::pgcd () {
  int a = _n, b = _d;
  int reste, tampon;
  if (a < b){
    tampon = a;
    a = b;
    b = tampon;
  }
  while (b != 0){
    reste = a % b;
    a = b;
    b = reste;
  }
  return a;
}

void Rationnel::inverser () {
  int tmp = _n;
  _n = _d;
  _d = tmp;
}

void Rationnel::affiche () {
  cout << _n << "/" << _d << endl;
}

int main () {
  Rationnel a (1, 2);
  Rationnel b = a;
  b.inverser ();
  b.affiche ();
}
```

3.2.3 Encapsulation des données

Si on choisit de mettre en private les données d'une classe, cela nous laisse libre de choisir leur représentation. Cette représentation peut varier suivant l'utilisation qui sera faite de la classe, toutefois comme ces données sont privées les programmes écrit pour une nouvelle représentation resteront compatibles avec les anciens programmes.

Pour illustrer l'intérêt de l'encapsulation nous allons écrire une classe Complexe de deux façons.

En tête de la classe Complexe avec des coordonnées cartésiennes :

```
#ifndef COMPLEXE_H
#define COMPLEXE_H

const float PI = 3.141592654;

class Complexe {
  float _x, _y;
 public :
  Complexe (float x = 0.0, float y = 0.0);
  float abscisse ();
  float ordonnee ();
  float rho ();
  float theta ();
  void ajoute (const Complexe & p);
  void homothetie (float h);
  void rotation (float angle);
};
#endif
```

Corps de la classe Complexe avec des coordonnées cartésiennes :

```
#include <iostream>
#include <cmath>

#include "Complexe.h"

using namespace std;

Complexe::Complexe (float x, float y) {
  _x = x;
  _y = y;
}

float Complexe::abscisse () { return _x; }

float Complexe::ordonnee () { return _y; }

float Complexe::rho () { return sqrt (_x * _x + _y * _y); }

// atan renvoie un angle entre -pi/2 et pi/2
// la fonction angle renvoie l'angle entre -pi et pi

float Complexe::theta () {
  if (_x == 0.0)
    return 0.0;
  return atan (_y/_x);
}

void Complexe::ajoute (const Complexe & p) {
  _x += p._x;
```

```
  _y += p._y;
}

void Complexe::homothetie (float h) {
  _x *= h;
  _y *= h;
}

void Complexe::rotation (float angle) {
  float r = rho ();
  float t = theta () + angle;
  _x = r * cos (t);
  _y = r * sin (t);
}

int main () {
  Complexe * c = new Complexe (1, 0);
  c->rotation (PI/2);
  cout << c->abscisse () << " + i" << c->ordonnee () << endl;
}
```

Si on a un programme qui fait beaucoup d'ajouts de complexes et peu de rotations, la classe en coordonnées cartésiennes convient tout à fait. Toutefois si on veut faire un nouveau programme qui fait une utilisation intensive des rotations, les calculs de la classe Complexe cartésienne peuvent être simplifiés si on utilise des coordonnées polaires en données privées plutot que des coordonnées cartésiennes.

Toutefois on ne veut pas changer les méthodes de la classe Complexe de façon à garder la compatibilité avec les programmes précédents.

La classe Complexe devient alors :

```
#ifndef COMPLEXE_H
#define COMPLEXE_H

const float PI = 3.141592654;

class Complexe {
  float _r, _t;
 public :
  Complexe (float x = 0.0, float y = 0.0);
  float abscisse ();
  float ordonnee ();
  float rho ();
  float theta ();
  void ajoute (const Complexe & p);
  void homothetie (float h);
  void rotation (float angle);
};
#endif
```

Corps de la classe Complexe avec des coordonnées polaires :

```
#include <iostream>
#include <cmath>

#include "Complexe.h"

using namespace std;

Complexe::Complexe (float x, float y) {
  _r = sqrt (x * x + y * y);
  _t = atan (y / x);
}

float Complexe::abscisse () { return _r * cos (_t); }

float Complexe::ordonnee () { return _r * sin (_t); }

float Complexe::rho () { return _r; }

float Complexe::theta () { return _t; }

void Complexe::ajoute (const Complexe & p) {
  int x += abscisse () + p.abscisse ();
  int y += ordonnee () + p.ordonnee ();
  _r = sqrt (x * x + y * y);
  _t = atan (y / x);

}

void Complexe::homothetie (float h) {
  _r *= h;
}

void Complexe::rotation (float angle) {
  _t += angle;
}
```

3.2.4 Une classe pile dynamique

La classe vector de la STL permet d'utiliser des tableaux de taille variable, nous allons programmer une classe analogue pour les piles d'entiers.

```
#include <iostream>

using namespace std;

class Pile {
  int _tailleMax;
  int _taille;
  int * _tab;

 public :
```

```
  Pile (int n = 8) {
    _tailleMax = n;
    _tab = new int [_tailleMax];
    _taille = 0;
  }

  ~Pile () { delete [] _tab; }

  int size () { return _taille; }

  int top () {
    if (_taille > 0)
      return _tab [_taille - 1];
    else
      cerr << "erreur Pile::top ()\n";
    return 0;
  }

  void pop () {
    if (_taille > 0)
      _taille--;
    else
      cerr << "erreur Pile::pop ()\n";
  }

  void push (int e) {
    if (_taille == _tailleMax) {
      _tailleMax *= 2;
      int * t = new int [_tailleMax];
      for (int i = 0; i < _taille; i++)
        t [i] = _tab [i];
      delete [] _tab;
      _tab = t;
    }
    _tab [_taille] = e;
    _taille++;
  }

  void affiche () {
    cout << "[";
    for (int i = 0; i < _taille; i++)
cout << _tab [i] << ",";
    cout << "]\n";
  }
};

int main () {
  Pile * p = new Pile;

  for (int i = 0; i < 10; i++)
    p->push (i);

  p->affiche ();
```

```
  Pile *inverse = new Pile;

  while (p->size ()) {
    inverse->push (p->top ());
    p->pop ();
  }

  inverse->affiche ();
}
```

3.2.5 Une classe liste chainée

La classe list de la STL permet d'utiliser des liste chainées, nous allons programmer une classe analogue pour les entiers.

```
#include <iostream>

using namespace std;

class Node {
  public:
    int data;
    Node *next;
};


class Liste {
 private:
  Node *first;
  int _taille;

  void deleteListe () {
    Node * tmp = first, * precedent;
    while (tmp != NULL) {
      precedent = tmp;
      tmp = tmp->next;
      delete precedent;
    }
  }

  Node * copy (Node * source) {
    if (source == NULL)
      return NULL;
    Node * tete = new Node;
    Node * tmp = tete;
    tete->data = source->data;
    source = source->next;
    while (source != NULL) {
      tmp->next = new Node;
      tmp = tmp->next;
```

```
      tmp->data = source->data;
      source = source->next;
    }
    tmp->next = NULL;
    return tete;
  }

 public:

  Liste () {
    _taille = 0;
    first = NULL;
  }

  ~Liste() {
    deleteListe ();
  }

  Liste (const Liste & source) {
    _taille = source._taille;
    first = copy (source.first);
  }

  Liste & operator= (const Liste & s) {
    if (this != &s) {
      deleteListe ();
      _taille = s._taille;
      first = copy (s.first);
    }
    return *this;
  }

  bool empty() {
    return (first == NULL);
  }

  void insere (int item, unsigned pos) {
    if (pos > _taille + 1) {
      cout<<"Illegal position to insert:"<<pos<<endl;
      return;
    }
    _taille++;
    Node* newNode = new Node;
    newNode->data = item;
    Node* prev = first;
    for (int i = 1; i < pos; i++)
      prev = prev->next;
    newNode->next = prev->next;
    prev->next = newNode;
  }

  void ajoute (int item) {
    _taille++;
    Node* newNode = new Node;
```

```
    newNode->data = item;
    newNode->next = NULL;
    Node* prev = first;
    if (prev == NULL)
      first = newNode;
    else {
      while (prev->next != NULL)
        prev = prev->next;
      prev->next = newNode;
    }
  }

  void ote (int item) {
    _taille--;
    Node * tmp = first;
    Node * precedent;
    while (tmp != NULL && tempP->data != item) {
      precedent = tmp;
      tmp = tmp->next;
    }
    if (tmp != NULL && tempP->data == item) {
      precedent->next = tmp->next;
      delete tmp;
    }
    else
      cerr << "Item to delete not found" << endl;
  }

  int locate(int item) {
    int position = 0;
    Node* ptr = first;
    while(ptr->data != item && ptr != NULL) {
      position++;
      ptr = ptr->next;
    }
    return position;
  }

  void affiche () {
    Node * tmp = first;
    cout << "[";
    while (tmp != 0) {
      cout << tmp->data << ",";
      tmp = tmp->next;
    }
    cout << "]\n";
    return;
  }
};

int main () {
  Liste * l = new Liste;

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

```
    l->ajoute (i);

  l->affiche ();
}
```

3.2.6 Surcharge des opérateurs et fonctions amies

Les opérateurs suivants peuvent être surchargés (ils sont donnés par ordre de priorité) :

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

On peut déclarer la fonction de surcharge d'un opérateur comme une fonction membre ou comme une fonction amie d'une classe.

Une fonction amie externe à la classe est déclarée comme *friend* à l'intérieur de la classe. Son contenu peut ensuite être défini à l'extérieur de la classe. Les fonctions amies ont la particularité de pouvoir accéder aux données privées de la classe amie.

Nous allons maintenant redéfinir les opérations de base sur les nombres rationnels avec la surcharge d'opérateurs. Nous en profiterons aussi pour surcharger les opérateurs d'entrée sortie avec des fonctions amies.

```
#include <iostream>

using namespace std;

class Rationnel {
 private :
  int _n, _d;

 public :
  Rationnel (int n = 1, int d = 1) {
    _n = n;
    _d = d;
    simplification (pgcd ());
  }
```

```
int numerateur () { return _n; }

int denominateur () { return _d; }

void simplification (int p) {
  _n /= p;
  _d /= p;
}

int pgcd () {
  int a = _n, b = _d;
  int reste, tampon;
  if (a < b){
    tampon = a;
    a = b;
    b = tampon;
  }
  while (b != 0){
    reste = a % b;
    a = b;
    b = reste;
  }
  return a;
}

void inverser () {
  int tmp = _n;
  _n = _d;
  _d = tmp;
}

void affiche () {
  cout << _n << "/" << _d << endl;
}

Rationnel operator + (Rationnel & r) {
  Rationnel res;
  res._d = _d * r._d;
  res._n = _n * r._d + r._n * _d;
  res.simplification (res.pgcd ());
  return res;
}

Rationnel operator - (Rationnel & r) {
  Rationnel res;
  res._d = _d * r._d;
  res._n = _n * r._d - r._n * _d;
  res.simplification (res.pgcd ());
  return res;
}

Rationnel operator * (Rationnel & r) {
  Rationnel res;
```

```
    res._n = _n * r._n;
    res._d = _d * r._d;
    res.simplification (res.pgcd ());
    return res;
  }

  Rationnel operator / (Rationnel & r) {
    Rationnel res;

    res._n = _n * r._d;
    res._d = _d * r._n;
    res.simplification (res.pgcd ());
    return res;
  }

  friend ostream & operator << (ostream &, Rationnel);
  friend istream & operator >> (istream &, Rationnel &);
};

ostream & operator << (ostream & sortie, Rationnel r) {
  sortie << "(" << r._n << "/" << r._d << ")";
  return sortie;
}

istream & operator >> (istream & entree, Rationnel & r) {
  entree >> r._n;
  entree >> r._d;
  return entree;
}

int main () {
  Rationnel a (1, 2);
  Rationnel b = Rationnel (1) / a;
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "a + b = " << a + b << endl;
  cout << "a - b = " << a - b << endl;
  cout << "a * b = " << a * b << endl;
  cout << "a / b = " << a / b << endl;
  cout << "Entrez un rationnel c " << endl;
  Rationnel c;
  cin >> c;
  cout << "c = " << c << endl;
  cout << "a + c = " << a + c << endl;
  cout << "a - c = " << a - c << endl;
  cout << "a * c = " << a * c << endl;
  cout << "a / c = " << a / c << endl;

}

/*
a = (1/2)
b = (2/1)
```

```
a + b = (5/2)
a - b = (3/-2)
a * b = (1/1)
a / b = (1/4)
Entrez un rationnel c
3
4
c = (3/4)
a + c = (5/4)
a - c = (1/-4)
a * c = (3/8)
a / c = (2/3)
*/
```

Un autre exemple est la surcharge de l'opérateur [] pour le type liste que l'on a vu précédemment. Il peut s'écrire comme une fonction membre :

```
int Liste::operator [] (int i) {
  int position=0;
  Node* ptr=first;
  while(position < i && ptr != NULL) {
    position++;
    ptr=ptr->next;
  }
  return ptr->data;
}
```

De même la fonction affiche peut être avantageusement remplacée par une surcharge de l'opérateur « :

```
ostream & operator << (ostream & sortie, Liste l) {
  Node * tmp = l.first;
  sortie << "[";
  while (tmp != NULL) {
    sortie << tmp->data << ",";
    tmp = tmp->next;
  }
  sortie << "]\n";
  return sortie;
}
```

Chapitre 4

Contrôle du 11 Mars 2010

On souhaite réaliser des opérations sur des vecteurs et des matrices.

4.1 La classe Vecteur

Les données de la classe *Vecteur* sont un pointeur *tab* sur un tableau d'entiers ainsi qu'un entier *_taille* qui donne la taille du tableau. Ecrire la classe Vecteur comportant les méthodes suivantes :

1. Le constructeur `Vecteur (int taille)` qui alloue la mémoire nécessaire au tableau *tab* et affecte *_taille*. Les valeurs du tableau seront tirées aléatoirement.
2. Le destructeur.
3. L'opérateur + qui additionne deux vecteurs de même taille et retourne un vecteur.
4. L'opérateur * qui fait le produit scalaire de deux vecteurs de même taille et retourne un entier. Si le vecteur V=$\{v_1, v_2, ..., v_n\}$ et le vecteur U=$\{u_1, u_2, ..., u_n\}$ alors le produit scalaire vaut $\sum_{j=1}^{j=n} v_j \times u_j$.
5. L'opérateur << qui affiche un vecteur.
6. Une méthode *void ecrit (const char * nomFichier)* qui écrit un vecteur dans le fichier *nomFichier*.
7. Une méthode *void lit (const char * nomFichier)* qui lit un vecteur dans le fichier *nomFichier*.

4.2 La classe Matrice

On demande d'écrire une classe *Matrice* pour les matrices carrées. La taille maximum d'une matrice est *TailleMax=100*. Ecrire une classe Matrice comportant les méthodes suivantes :

1. Le constructeur `Matrice (int taille)`. Les valeurs de la matrice seront tirées aléatoirement.
2. L'opérateur * qui multiplie une matrice par un vecteur, le résultat de l'opération étant un vecteur. Si l'élément de la rangée i et de la colonne j de la matrice A est a_{ij}, que le vecteur V=$\{v_1, v_2, ..., v_n\}$. Alors le produit A * V donne le vecteur U=$\{u_1, u_2, ..., u_n\}$ tel que $u_i = \sum_{j=1}^{j=n} a_{ij} \times v_j$.
3. Ecrire de plus une fonction *main ()* qui déclare un vecteur et une matrice et affiche le resultat de la multiplication de la matrice par le vecteur.

4.3 Correction

```
#include<iostream>
#include<fstream>
#include<stdlib.h>

using namespace std;

class Vecteur {
 public:
  int _taille, *tab;

  Vecteur (int taille) {
    _taille = taille;
    tab = new int [taille];
    for (int i = 0; i < taille; i++)
      tab [i] = rand () % 10;
  }

  ~Vecteur () {
    delete [] tab;
  }

  Vecteur & operator + (const Vecteur & v) {
    Vecteur * res = new Vecteur (_taille);
    for (int i = 0; i < _taille; i++)
      res->tab [i] = tab [i] + v.tab [i];
    return *res;
  }

  int operator * (const Vecteur & v) {
    int res = 0;
    for (int i = 0; i < _taille; i++)
      res += tab [i] * v.tab [i];
    return res;
  }

  Vecteur & operator = (const Vecteur & v) {
    delete [] tab;
    tab = new int [v._taille];
    _taille = v._taille;
    for (int i = 0; i < _taille; i++)
```

```
      tab [i] = v.tab [i];
    return *this;
  }

  friend ostream & operator << (ostream & sortie, const Vecteur & v);

  void ecrit (const char * nomFichier) {
    ofstream sortie (nomFichier, ios::out);
    sortie << _taille << " ";
    for (int i = 0; i < _taille; i++)
      sortie << tab [i] << " ";
    sortie.close ();
  }

  void lit (const char * nomFichier) {
    ifstream entree (nomFichier, ios::in);
    entree >> _taille;
    delete [] tab;
    tab = new int [_taille];
    for (int i = 0; i < _taille; i++)
      entree >> tab [i];
    entree.close ();
  }
};

ostream & operator << (ostream & sortie, const Vecteur & v) {
  sortie << "[";
  for (int i = 0; i < v._taille; i++) {
    sortie << v.tab [i];
    if (i < v._taille - 1)
      sortie << ",";
  }
  sortie << "]";
  return sortie;
}

const int TailleMax = 100;

class Matrice {
 public:
  int _taille;
  int mat [TailleMax] [TailleMax];

  Matrice (int taille) {
    _taille = taille;
    for (int i = 0; i < taille; i++)
      for (int j = 0; j < taille; j++)
        mat [i] [j] = rand () % 2;
  }

  Vecteur & operator * (const Vecteur & v) {
    Vecteur * res = new Vecteur (_taille);

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

```
      res->tab [i] = 0;
      for (int j = 0; j < _taille; j++)
        res->tab [i] += mat [i] [j] * v.tab [j];
    }
    return *res;
  }
  friend ostream & operator << (ostream & sortie, const Matrice & m);
};

ostream & operator << (ostream & sortie, const Matrice & m) {
  for (int i = 0; i < m._taille; i++) {
    for (int j = 0; j < m._taille; j++) {
      sortie << m.mat [i] [j] << " ";
    }
    sortie << endl;
  }
  return sortie;
}

int main () {
  Vecteur v (10), u (10);
  Matrice m (10);

  u = m * v;
  cout << m << endl;
  cout << v << endl;
  cout << u << endl;
  u.ecrit ("toto");
  u.lit ("toto");
  cout << u << endl;
}
```

Chapitre 5

L'héritage

L'héritage permet d'étendre une classe existante en lui ajoutant des données et des méthodes.

Une classe dérivée est un cas particulier d'une classe de base, elle reprend ses données ainsi que ses méthodes publiques.

Pour expliquer comment on utilise l'héritage, nous prendrons comme exemple une simulation multi-agents.

5.1 L'héritage simple

Les agents de la simulation ont tous en commun un emplacement. La classe de base Agent contiendra donc deux entiers qui précisent les coordonnées de cet emplacement :

```
class Agent {
  int _x, _y;

 public:

  Agent (int x = 0, int y = 0) {
    _x = x;
    _y = y;
  }

  void set (int x, int y) {
    _x = x;
    _y = y;
  }

  void affiche () {
    cout << "(" << _x << "," << _y << ")" << endl;
  }
```

```
};
```

5.1.1 Classe dérivée

On suppose qu'on est dans un jeu de stratégie temps réél et que les agents peuvent se déplacer ou non. On veut ajouter des agents qui se déplacent vers un but fixé. On étend donc la classe Agent vers une nouvelle classe AgentMobile qui contiendra l'emplacement objectif de l'agent en plus de sa position. La classe AgentMobile dérive de la classe Agent :

```
class AgentMobile : public Agent {
  int _xBut, _yBut;

 public:
  AgentMobile (int x, int y, int xBut, int yBut) {
    set (x, y);
    _xBut = xBut;
    _yBut = yBut;
  }
};
```

La classe AgentMobile hérite des données privées _x et _y de Agent, toutefois comme ces données sont privées dans la classe Agent, la classe AgentMobile n'y a pas accès. Elle doit faire appel à la méthode set qui est publique pour les modifier.

Si l'on veut que les classe dérivées puissent avoir accès à certaines données d'une classe de base, il faut déclarer ces données protected :

```
class Agent {
 protected:
  int _x, _y;

 public:

  Agent (int x = 0, int y = 0) {
    _x = x;
    _y = y;
  }
};
```

De cette façon, la classe AgentMobile aura accès aux variables _x et _y mais les classes qui ne sont pas des classes dérivées de Agent n'y auront pas accès.

5.1.2 Constructeur et destructeur

Lorsqu'on déclare un objet d'une classe dérivée :

```
AgentMobile toto (0, 0, 10, 10);
```

C++ commence par allouer la place pour les données de Agent et les données de AgentMobile, puis il appelle le constructeur de Agent, et ensuite le constructeur de AgentMobile.

Lorsqu'on détruit un objet d'une classe dérivée, on commence par appeler le destructeur des méthodes dérivées avant ceux des classe de bases. Ici on appelle le destructeur de AgentMobile avant celui de Agent et ceci avant de libérer la zone mémoire contenant les données.

Si on ne veut pas utiliser le constructeur par défaut pour la classe de base, il faut appeler le constructeur voulu dans la liste d'initialisation :

```
class AgentMobile : public Agent {
  int _xBut, _yBut;

 public:

  AgentMobile (int x, int y, int xBut, int yBut) : Agent (x, y) {
    _xBut = xBut;
    _yBut = yBut;
  }
};
```

5.1.3 Fonctions membres

On peut redéfinir dans une classe dérivée un méthode qui existe déjà dans une classe de base. Par exemple on peut redéfinir l'affichage d'un AgentMobile :

```
class AgentMobile : public Agent {
  int _xBut, _yBut;

 public:
  AgentMobile (int x, int y, int xBut, int yBut) : Agent (x, y) {
    _xBut = xBut;
    _yBut = yBut;
  }

  void affiche () {
    cout << "(" << _x << "," << _y << "," << _xBut << "," << _yBut << ")" << endl;

  }
};
```

Si on écrit le programme suivant :

```
void main () {
  Agent toto (5, 5);
  AgentMobile titi (3, 3, 10, 10);

  toto.affiche ();
```

```
  titi.affiche ();
}
```

Il affichera :

```
(5,5)
(3,3,10,10)
```

La méthode affiche () de la classe dérivée AgentMobile masque la méthode affiche () de la classe de base.

Il est aussi possible d'appeler un fonction qui prend en paramètre un objet de la classe de base avec un objet de la classe dérivée :

```
void afficheAgent (Agent & a) {
  a.affiche ();
}

void main () {
  Agent toto (5,5);
  AgentMobile titi (3, 3, 10,10);

  afficheAgent (toto);
  afficheAgent (titi);
}
```

Cette fois, le programme affichera :

```
(5,5)
(3,3)
```

5.2 Les fonctions virtuelles

Nous venons de voir que la fonction membre appelée pour un objet correspond au typage statique de l'objet : on appelle la fonction membre de la classe avec laquelle on a déclaré l'objet même si l'objet est en fait un objet d'une classe dérivée.

Les fonctions virtuelles permettent d'utiliser un typage dynamique au lieu du typage statique qui est détéminé à la compilation.

Dans notre exemple, pour que l'affichage affiche le bon objet même s'il n'a pas le bon type, on déclare la fonction affiche () comme virtual :

```
#include<iostream>

using namespace std;
```

```
class Agent {
 protected:
  int _x, _y;

 public :

  Agent (int x = 0, int y = 0) {
    _x = x;
    _y = y;
  }

  virtual void affiche () {
    cout << "(" << _x << "," << _y << ")" << endl;
  }
};

class AgentMobile : public Agent {
  int _xBut, _yBut;

 public:

  AgentMobile (int x, int y, int xBut, int yBut) : Agent (x, y) {
    _xBut = xBut;
    _yBut = yBut;
  }

  virtual void affiche () {
    cout << "(" << _x << "," << _y << "," << _xBut << "," << _yBut << ")" << endl;

  }
};

void afficheAgent (Agent & a) {
  a.affiche ();
}

int main () {
  Agent toto (5,5);
  AgentMobile titi (3, 3, 10,10);

  afficheAgent (toto);
  afficheAgent (titi);

  Agent tata = titi;
  afficheAgent (tata);
}
```

le programme affiche :

```
(5,5)
(3,3,10,10)
(3,3)
```

Déclarer une méthode virtuelle permet d'appeler la méthode de la classe de l'objet lorsqu'il a été créé et non la méthode de la classe de son type actuel.

L'appel de méthodes différentes suivant la classe de l'objet s'appelle le polymorphisme dynamique. Le polymorphisme ne peut pas être utilisé pour choisir dynamiquement une méthode à appeler d'après le type de son paramètre :

```
void afficheAgent (Agent & a) {
  a.affiche ();
}

void afficheAgent (AgentMobile & a) {
  a.affiche ();
}

int main () {
  Agent * a = new AgentMobile;
  afficheAgent (*a);
}
```

Dans cet exemple c'est la première méthode qui est appelée alors que a est en fait un AgentMobile.

Toutefois il peut être utilisé pour s'assurer que l'appel au destructeur fait toujours appel au destructeur de l'objet qui a été initialement alloué. On déclare alors le destructeur de la façon suivante :

```
class Agent {
 public:
  virtual void ~Agent () {};
};
```

5.2.1 Méthode virtuelle pure

Supposons qu'on veuille maintenant que toutes les classes d'agents aient une méthode affiche (). Pour forcer cette méthode à être présente, on utilise une méthode virtuelle pure. On définit par exemple pour cela une classe ObjetAffichable dans laquelle on déclare une méthode affiche () virutelle pure et on en fait dériver tous les agents :

```
#include<iostream>

using namespace std;

class ObjetAffichable {
  virtual void affiche () = 0;
}

class Agent : public ObjetAffichable {
 protected:
```

```
  int _x, _y;

 public :

  Agent (int x = 0, int y = 0) {
    _x = x;
    _y = y;
  }

  virtual void affiche () {
    cout << "(" << _x << "," << _y << ")" << endl;
  }
};
```

La classe ObjetAffichable est abstraite : elle ne peut pas être instanciée.

5.3 L'héritage multiple

On peut hériter de plusieurs classes de base. Par exemple si on a des classes qui décrivent des obstacles et que certains obstacles sont mobiles alors que d'autres sont immobiles. On pourra décrire cela avec un héritage multiple :

```
#include<iostream>

using namespace std;

class ObjetAffichable {
  virtual void affiche () = 0;
};

class Agent : public ObjetAffichable {

 protected:
  int _x, _y;

 public :

  Agent (int x, int y) {
    _x = x;
    _y = y;
  }

  virtual void affiche () {
    cout << "(" << _x << "," << _y << ")" << endl;
  }
};

class AgentMobile : virtual public Agent {
 protected:
  int _xBut, _yBut;
```

```
 public:

  AgentMobile (int x, int y, int xBut, int yBut) : Agent (x, y) {
    _xBut = xBut;
    _yBut = yBut;
  }

  virtual void affiche () {
    cout << "(" << _x << "," << _y << "," << _xBut << "," << _yBut << ")" << endl;

  }
};

class Obstacle : virtual public Agent {
 protected:
  bool _destructible;
 public:
  Obstacle (int x, int y, bool d = true) : Agent (x, y) {
    _destructible = d;
  }
};

class ObstacleMobile : public Obstacle, public AgentMobile {
 public :

  ObstacleMobile (int x, int y, int xBut, int yBut, bool d) :
    Agent (x, y), AgentMobile (x, y, xBut, yBut), Obstacle (x, y, d) {}

  virtual void affiche () {
    cout << "(" << _x << "," << _y << "," << _xBut << "," << _yBut << "," << _destructible

  }
};

void afficheAgent (Agent & a) {
  a.affiche ();
}

int main () {
  ObstacleMobile o (3, 3, 10, 10, true);
  afficheAgent (o);
}
```

On voit qu'on a utilisé un héritage virtuel pour la classe Agent car sinon celle ci se serait retrouvée deux fois dans la classe ObstacleMobile (une fois par Obstacle et une fois par AgentMobile). Utiliser un héritage virtuel permet à ses données de n'être présentes qu'une seule fois dans la classe dérivée.

Chapitre 6

TD 2

6.1 Courbes fractales

Ecrire la classe Complexe avec surcharge des opérateurs + et *.

L'ensemble de Julia est défini pour les points du plan par l'ensemble des points pour lesquels la suite $z_{n+1} = z_n^2 + c$ ne diverge pas.

z_0 parcourt au début tous les points contenus dans un carré centré sur l'origine, chaque point correspondant à un complexe. On peut par exemple choisir pour $c = -0.05 + i \times 0.7$.

Ecrire un programme qui calcule un ensemble de Julia.

L'ensemble de Mandelbrot est défini par l'ensemble des points c tels que $z_{n+1} = z_n^2 + c$ ne diverge pas avec $z_0 = 0$.

Modifier le programme pour qu'il calcule l'ensemble de Mandelbrot.

6.2 Classe Expression

On définit la classe Expression de la façon suivante :

```
class Expression {
 public:
  virtual float evalue () = 0;
  virtual void affiche () = 0;
}
```

Ecrire les classes pour les opérations de base et pour les réels. On engendre aléatoirement des expressions dont on calcule le résultat.

6.3 Simulation Multi-Agents

On implémentera les classes agents vues en cours en faisant la distinction entre agents mobiles et immobiles et entre obstacles mobiles et immobiles. Lorsqu'un agent rencontre un obstacle destructible, celui ci est détruit.

On fera une simulation sur une carte 10x10 d'agents et d'obstacles se déplaçant aléatoirement, la simulation se terminant quant tous les obstacles destructibles ont été détruits.

On pourra par la suite améliorer les agents mobiles en leur programmant une stratégie de déplacement.

6.4 Algorithme générique de recherche

Ecrire une classe générique qui stocke des éléments avec un arbre binaire de recherche. Les éléments doivent pouvoir être comparé avec les opérateurs $>$ et $=$. On ecrira une fonction d'ajout d'un élément, un fonction de suppression d'un élément et une fonction de recherche d'un élément.

On utilisera la classe pour deux types de données : les entiers et les rationnels.

On écrira ensuite une classe qui a les mêmes fonctionnalités à l'aide de la STL.

Chapitre 7

Les templates

Les templates permettent d'utiliser les classes ou les algorithmes avec une classe fournie en paramètre. La classe utilisée en paramètre étant modifiable, on peut appliquer l'algorithme à autant de classes différentes que l'on veut sans le changer.

Par exemple une classe liste paramétrée pourra être utilisée aussi bien pour créer des listes d'entiers que des listes d'agents ou des listes de pommes.

Les templates permettent d'écrire des algorithmes et des classes génériques. Par exemple un algorithme générique de tri peut être défini par des templates en faisant l'hypothèse que chaque classe passée en paramètre a une fonction de comparaison des objets.

7.1 Algorithmes génériques

Pour écrire un algorithme générique on utilise le mot-clé template suivi de la classe paramétrée :

```
template <class T>
T minimum (T x, T y) {
  if (x > y)
    return y;
  return x;
}
```

Un fois cette fonction définie, on peut l'utiliser avec les types pour lesquels l'opérateur $>$ est défini. Par exemple :

```
int x = 3, y = 10, z = minimum (x, y);
float f = 3.0, g = 6.7, h = minimum (f, g);
```

Le compilateur construira deux fonctions maximum, une pour les entiers et une pour les réels.

Si on suppose que la classe paramétrée contient l'opérateur $>$, on peut aussi écrire un algorithme générique de tri d'un tableau d'objets :

```
template <class T>
void tri (T *tab, int taille) {
  for (int i = 0; i < taille; i++)
    for (int j = i + 1; j < taille; j++)
       if (tab [i] > tab [j]) {
         T temp = tab [i];
         tab [i] = tab [j];
         tab [j] = temp;
       }
}
```

On peut alors appliquer cet algorithme pour le tri des points du plan en fonction de leur distance à l'origine :

```
#include <iostream>
#include <stdlib.h>

using namespace std;

class Point {
 public :
  float _x, _y;

  Point () {
    _x = rand () / (RAND_MAX + 1.0);
    _y = rand () / (RAND_MAX + 1.0);
  }

  bool operator > (const Point & p) {
    return _x * _x + _y * _y > p._x * p._x + p._y * p._y;
  }
  friend ostream & operator << (ostream &, Point);
};

ostream & operator << (ostream & sortie, Point p) {
  sortie << "(" << p._x << "," << p._y << ")";
  return sortie;
}

template <class T>
void tri (T *tab, int taille) {
  for (int i = 0; i < taille; i++)
    for (int j = i + 1; j < taille; j++)
       if (tab [i] > tab [j]) {
         T temp = tab [i];
         tab [i] = tab [j];
         tab [j] = temp;
       }
}
```

```
int main () {
  Point tab [10];
  tri (tab, 10);
  for (int i = 0; i < 10; i++)
    cout << tab [i] << endl;
  return 0;
}
```

7.2 Définition d'une classe générique

On peut aussi définir des classes qui prennent en paramètre des classes. Par exemple si on veut utiliser un pile de taille variable pour n'importe quel type ou classe on écrit :

```
#include <iostream>
#include <stdlib.h>

using namespace std;

template <class T>
class Pile {
  T * _tab;
  int _size, _maxSize;

 public :

  Pile () {
    _size = 0;
    _maxSize = 2;
    _tab = new T [_maxSize];
  }

  void push (const T & e) {
    if (_size >= _maxSize) {
      T * t = new T [_maxSize * 2];
      for (int i = 0; i < _size; i++)
         t [i] = _tab [i];
      delete [] _tab;
      _tab = t;
      _maxSize = _maxSize * 2;
    }
    _tab [_size] = e;
    _size++;
  }

  T top () {
    if (_size > 0)
      return _tab [_size - 1];
    else {
      cerr << "erreur Pile::top\n";
```

```
      return _tab [0];
    }
  }

  void pop () {
    if (_size > 0)
      _size--;
  }
};

int main () {
  Pile<int> p;

  for (int i = 0; i < 10; i++)
    p.push (rand ());

  for (int i = 0; i < 10; i++) {
    cout << p.top () << endl;
    p.pop ();
  }

  return 0;
}
```

On peut aussi passer plusieurs paramètres à la classe générique, par exemple pour définir explicitement la taille maximum du tableau on peut écrire :

```
#include <iostream>
#include <stdlib.h>

using namespace std;

template <class T, int Max>
class PileMax {
  T _tab [Max];
  int _size;

 public :

  PileMax () {
    _size = 0;
  }

  void push (const T & e) {
    if (_size >= Max) {
      cerr << "erreur PileMax::push\n";
      return;
    }
    _tab [_size] = e;
    _size++;
  }

  T top () {
```

```
    if (_size > 0)
      return _tab [_size - 1];
    else {
      cerr << "erreur PileMax::top\n";
      return _tab [0];
    }
  }

  void pop () {
    if (_size > 0)
      _size--;
  }
};

int main () {
  PileMax<int, 100> p;

  for (int i = 0; i < 10; i++)
    p.push (rand ());

  for (int i = 0; i < 10; i++) {
    cout << p.top () << endl;
    p.pop ();
  }

  return 0;
}
```

On peut aussi spécialiser une fonction pour un paramètre générique. Par exemple pour l'affichage d'une pile de réels, on veut utiliser une précision de quatre chiffres alors qu'on utilise l'affichage normal pour les autres types, on écrira alors :

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>

using namespace std;

template <class T>
class Pile {
  T * _tab;
  int _size, _maxSize;

 public :

  Pile ();
  void push (const T & e);
  T top ();
  void pop ();
  void affiche ();
};

template <class T> void Pile<T>::affiche () {
```

```
  for (int i = 0; i < _size; i++)
    cout << _tab [i] << endl;
}

template <> void Pile<float>::affiche () {
  for (int i = 0; i < _size; i++)
    cout << setprecision (4) << _tab [i] << endl;
}

int main () {
  Pile<int> p;

  for (int i = 0; i < 3; i++)
    p.push (rand ());

  p.affiche ();

  Pile<float> p1;

  for (int i = 0; i < 3; i++)
    p1.push (rand ());

  p1.affiche ();

  return 0;
}
```

Pour spécialiser une classe entière on utilisera :

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>

using namespace std;

template <class T>
class Pile {
  T * _tab;
  int _size, _maxSize;

 public :

  Pile () {
    _size = 0;
    _maxSize = 2;
    _tab = new T [_maxSize];
  }
  void push (const T & e);
  T top ();
  void pop ();
  void affiche ();
};
```

```
template <> class Pile<float> {
  float * _tab;
  int _size, _maxSize;

 public :

  Pile ();
  void push (const float & e);
  float top ();
  void pop ();
  void affiche ();
};

Pile<float>::Pile () {
  _size = 0;
  _maxSize = 2;
  _tab = new float [_maxSize];
}

int main () {
  Pile<int> p;
  Pile<float> p1;
}
```

On peut aussi transmettre des paramètres par défaut à un patron :

```
#include <iostream>
#include <stdlib.h>

using namespace std;

template <class T = Complexe, int Max = 100>
class PileMax {
  T _tab [Max];
  int _size;

 public :

  PileMax () {
    _size = 0;
  }
};

int main () {
  PileMax<> p;
  PileMax<int> p1;
  PileMax<float, 1000> p2;
}
```

7.3 La STL

La STL est la Standard Template Library. C'est un ensemble d'algorithmes et de classes génériques que l'on peut utiliser pour de nombreux types de données.

Les conteneurs sont des classes génériques qui représentent des structures de données couramment utilisées. Par exemple *list* permet d'avoir une implémentation générique des listes, *vector* des tableaux à taille variable, *stack* des piles, *queue* des files, *priority_ queue* des files de priorité, *set* des éléments triés par un arbre binaire de recherche, *deque* des files pour lesquelles on peut ajouter des éléments aussi bien au début qu'à la fin, *map* qui permet de stocker des éléments associés à une clé, etc.

Pour ces classes on dispose d'itérateurs qui sont une généralisation des pointeurs qui permettent de parcourir les objets d'une classe.

Les itérateurs sont utilisés par la bibliothèque d'algorithmes génériques de la STL.

Par exemple pour utiliser un tableau dynamique d'entiers on écrit :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdlib.h>

using namespace std;

bool comparateur (int i, int j) {
  return (i<j);
}

struct compare {
  bool operator() (int i, int j) { return (i<j); }
} compare;

int main () {
  vector<int> v;

  for (int i = 0; i < 100; i++)
    v.push_back (rand ());

  sort (v.begin(), v.end(), comparateur);

  for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << *it << " ";
  cout << endl;

  v.clear ();

  for (int i = 0; i < 100; i++)
    v.push_back (rand ());

  sort (v.begin(), v.end(), compare);
```

```
  for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << *it << " ";
  cout << endl;

  return 0;
}
```

7.4 La classe vector

Pour utiliser des tableaux dynamiques d'entiers on utilise la classe vector. Elle peut être utilisée avec divers itérateurs :

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

main()
{
   vector<string> SS;

   SS.push_back("The number is 10");
   SS.push_back("The number is 20");
   SS.push_back("The number is 30");

   cout << "Loop by index:" << endl;

   int ii;
   for(ii=0; ii < SS.size(); ii++)
   {
      cout << SS[ii] << endl;
   }

   cout << endl << "Constant Iterator:" << endl;

   vector<string>::const_iterator cii;
   for(cii=SS.begin(); cii!=SS.end(); cii++)
   {
      cout << *cii << endl;
   }

   cout << endl << "Reverse Iterator:" << endl;

   vector<string>::reverse_iterator rii;
   for(rii=SS.rbegin(); rii!=SS.rend(); ++rii)
   {
      cout << *rii << endl;
   }
```

```
    cout << endl << "Sample Output:" << endl;

    cout << SS.size() << endl;
    cout << SS[2] << endl;

    swap(SS[0], SS[2]);
    cout << SS[2] << endl;
}
```

On peut aussi l'utiliser pour des tableaux à plusieurs dimensions :

```
#include <iostream>
#include <vector>

using namespace std;

main()
{
   // Declare size of two dimensional array and initialize.
   vector< vector<int> > vI2Matrix(3, vector<int>(2,0));

   vI2Matrix[0][0] = 0;
   vI2Matrix[0][1] = 1;
   vI2Matrix[1][0] = 10;
   vI2Matrix[1][1] = 11;
   vI2Matrix[2][0] = 20;
   vI2Matrix[2][1] = 21;

   cout << "Loop by index:" << endl;

   int ii, jj;
   for(ii=0; ii < 3; ii++)
   {
      for(jj=0; jj < 2; jj++)
      {
         cout << vI2Matrix[ii][jj] << endl;
      }
   }
}
```

Pour utiliser les itérateurs sur des vecteurs à deux dimension, on utilise :

```
#include <iostream>
#include <vector>

using namespace std;

main()
{
   vector< vector<int> > vI2Matrix;    // Declare two dimensional array
   vector<int> A, B;
```

```
   vector< vector<int> >::iterator iter_ii;
   vector<int>::iterator                iter_jj;

   A.push_back(10);
   A.push_back(20);
   A.push_back(30);
   B.push_back(100);
   B.push_back(200);
   B.push_back(300);

   vI2Matrix.push_back(A);
   vI2Matrix.push_back(B);

   cout << endl << "Using Iterator:" << endl;

   for(iter_ii=vI2Matrix.begin(); iter_ii!=vI2Matrix.end(); iter_ii++)
   {
      for(iter_jj=(*iter_ii).begin(); iter_jj!=(*iter_ii).end(); iter_jj++)
      {
         cout << *iter_jj << endl;
      }
   }
}
```

7.5 La classe list

La classe list représente des listes d'objets, voici un exemple pour les listes d'entiers :

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

// Simple example uses type int

main()
{
   list<int> L;
   L.push_back(0);              // Insert a new element at the end
   L.push_front(0);             // Insert a new element at the beginning
   L.insert(++L.begin(),2);     // Insert "2" before position of first argument
                                // (Place before second argument)
   L.push_back(5);
   L.push_back(6);

   list<int>::iterator i;

   for(i=L.begin(); i != L.end(); ++i) cout << *i << " ";
   cout << endl;
   return 0;
}
```

7.6 La classe set

Une objet set est une liste triée d'éléments. Il se déclare comme set<clé, fonction de comparaison>.

Les objets stockés sont de type clé. Un set ne peut pas contenir deux éléments identiques.

Nous allons montrer comment utiliser l'algorithme copy avec des set :

```
#include <iostream>  // pour cout
#include <algorithm> // pour copy
#include <set>
#include <string>
#include <iterator>  // pour ostream_iterator
using namespace std;

int main()
{
  set<char> unensemble;

  unensemble.insert('a');
  unensemble.insert('b');
  unensemble.insert('c');

  cout << "mon ensemble contient :" << endl;
  for (set<char>::iterator i(unensemble.begin());
       i != unensemble.end(); ++i) {
    cout << "    " << *i << endl;
  }

  string s;

  // pour utiliser copy, il est imperatif que la chaine ait
  // la place necessaire, donc ici au moins 3 caracteres.
  // En pratique on n'utilisera donc pas copy pour CREER un
  // nouveau container, mais bien pour COPIER des valeurs;
  s = "xxx";

  copy(unensemble.begin(), unensemble.end(), s.begin());

  cout << s << endl;

  // affichage
  cout << "mon ensemble = ";
  copy(unensemble.begin(), unensemble.end(),
       ostream_iterator<char>(cout, ", "));
  cout << endl;

  return 0;
}
```

Nous allons redéfinir des fonctions sur les set : Union, Intersection et Difference.

```
#include <iostream>
#include <set>
#include <string.h>

using namespace std;

struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

template <class InputIterator1, class InputIterator2, class Set>
  Set & Union ( InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                Set & result) {
  while (true)
  {
    if ((first1==last1) && (first2==last2)) return result;
    else if (first1==last1) result.insert (*first2++);
    else if (first2==last2) result.insert (*first1++);
    else if (*first1<*first2) result.insert (*first1++);
    else if (*first2<*first1) result.insert (*first2++);
    else {  result.insert (*first1++); first2++; }

  }
}

template <class InputIterator1, class InputIterator2, class Set>
  Set & Intersection (const Set & set1,  InputIterator1 first1,
                      const Set & set2, InputIterator2 first2,
                      Set & result) {
  InputIterator1 it1 = set1.begin ();
  while (true) {
    if (it1 == set1.end ())
      return result;
    else {
      InputIterator2 it2 = set2.find (*it1);
      if (it2 != set2.end ())
result.insert (*it1);
      it1++;
    }
  }
}

template <class InputIterator1, class InputIterator2, class Set>
  Set & Difference (const Set & set1,  InputIterator1 first1,
                    const Set & set2, InputIterator2 first2,
                    Set & result) {
```

```
  InputIterator1 it1 = set1.begin ();
  while (true) {
    if (it1 == set1.end ())
      return result;
    else {
      InputIterator2 it2 = set2.find (*it1);
      if (it2 == set2.end ())
result.insert (*it1);
      it1++;
    }
  }
}

int main() {
  const int N = 6;
  const char* a[N] = {"marc", "jean", "patrick",
                      "ernest", "david", "jerome"};
  const char* b[N] = {"david", "tristan", "ernest",
                      "olivier", "denis", "abdallah"};

  set<const char*, ltstr> A(a, a + N);
  set<const char*, ltstr> B(b, b + N);
  set<const char*, ltstr> C;

  cout << "Set A: ";
  for (set<const char*, ltstr>::iterator it = A.begin();
       it != A.end (); ++it)
    cout << *it << " ";
  cout << endl;
  cout << "Set B: ";
  for (set<const char*, ltstr>::iterator it = B.begin();
       it != B.end (); ++it)
    cout << *it << " ";
  cout << endl;

  cout << "Union: ";
  Union(A.begin(), A.end(), B.begin(), B.end(), C);
  for (set<const char*, ltstr>::iterator it = C.begin();
       it != C.end (); ++it)
    cout << *it << " ";
  cout << endl;

  cout << "Intersection: ";
  C.clear ();
  Intersection(A, A.begin (), B, B.begin (), C);
  for (set<const char*, ltstr>::iterator it = C.begin();
       it != C.end (); ++it)
    cout << *it << " ";
  cout << endl;

  C.clear ();
  Difference(A, A.begin(), B, B.begin(), C);
  cout << "Set C (difference of A and B): ";
  for (set<const char*, ltstr>::iterator it = C.begin();
```

```
       it != C.end (); ++it)
    cout << *it << " ";
  cout << endl;
}
```

7.7 La classe map

La class map contient des paires d'objet, clé + données. Elle se déclare comme map<Key, Data, Compare>. Deux objets d'une map ont forcément des clés différentes.

```
#include <iostream>
#include <map>
#include <string.h>

using namespace std;

struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  map<const char*, int, ltstr> months;

  months["january"] = 31;
  months["february"] = 28;
  months["march"] = 31;
  months["april"] = 30;
  months["may"] = 31;
  months["june"] = 30;
  months["july"] = 31;
  months["august"] = 31;
  months["september"] = 30;
  months["october"] = 31;
  months["november"] = 30;
  months["december"] = 31;

  cout << "june -> " << months["june"] << endl;
  map<const char*, int, ltstr>::iterator cur  = months.find("june");
  map<const char*, int, ltstr>::iterator prev = cur;
  map<const char*, int, ltstr>::iterator next = cur;
  ++next;
  --prev;
  cout << "Previous (in alphabetical order) is " << (*prev).first << endl;
  cout << "Next (in alphabetical order) is " << (*next).first << endl;
}
```

7.8 La classe multimap

La classe multimap représente des clés associées à des données, toutefois plusieurs éléments peuvent avoir la même clé. Elle se déclare avec multimap<Key, Data, Compare, Alloc>.

```
#include <iostream>
#include <map>
#include <string.h>

using namespace std;

struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  multimap<const char*, int, ltstr> m;

  m.insert(pair<const char* const, int>("a", 1));
  m.insert(pair<const char* const, int>("c", 2));
  m.insert(pair<const char* const, int>("b", 3));
  m.insert(pair<const char* const, int>("b", 4));
  m.insert(pair<const char* const, int>("a", 5));
  m.insert(pair<const char* const, int>("b", 6));

  cout << "Number of elements with key a: " << m.count("a") << endl;
  cout << "Number of elements with key b: " << m.count("b") << endl;
  cout << "Number of elements with key c: " << m.count("c") << endl;

  cout << "Elements in m: " << endl;
  for (multimap<const char*, int, ltstr>::iterator it = m.begin();
       it != m.end();
       ++it)
   cout << "  [" << (*it).first << ", " << (*it).second << "]" << endl;
}
```

7.9 La classe hash_set

Le classe hash set stocke des clés. Un objet se déclare avec hash_set<Clé, fonctionDeHachage, CléEgale>. Deux objets contenus dans un hash_set ont des clés différentes (cléEgale renvoie faux). L'intérêt de stocker les éléments avec une fonction de hachage est de retouver ces éléments rapidement à l'aide d'un table de hachage.

En g++, la classe hash_set a été remplacée par la classe unordered_set.

7.10 L'algorithme copy

L'algorithme copy utilise des itérateurs pour copier le contenu d'un conteneur compris entre deux itérateurs vers un itérateur d'un autre conteneur.

Par exemple pour copier un vecteur d'entiers vers une liste d'entiers on utilise :

```
copy (v.begin (), v.end (), l.begin ());
```

7.11 L'algorithme generate

L'algorithme generate permet d'initialiser un conteneur avec les résultats d'une fonction. Pour cela on passe en argument la fonction d'initialisation :

```
generate (v.begin (), v.end (), fonction);
```

Exemple d'utilisation avec une fonction :

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>

using namespace std;

int valeur = 0;

int fonction () {
  valeur++;
  return valeur;
}

int main () {
  vector<int> v (20, 0);

  generate (v.begin (), v.end (), fonction);
```

```
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, ", "));
  cout << endl;

  return 0;
}
```

Si on veut utiliser la fonction avec des valeurs de départ différentes, on peut utiliser une classe fonction qui initialisera la valeur de départ dans le constructeur :

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>

using namespace std;

class fonction {
  int valeur;
 public:
  fonction (int n) { valeur = n; }
  int operator () () {
    valeur++;
    return valeur;
  }
};

int main () {
  vector<int> v (20, 0);

  generate (v.begin (), v.end (), fonction (10));
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, ", "));
  cout << endl;
  generate (v.begin (), v.end (), fonction (100));
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, ", "));
  cout << endl;

  return 0;
}
```

7.12 Algorithmes de recherche

L'algorithme find cherche une valeur entre deux itérateurs. L'algorithme search cherche une suite de valeurs représentée par un itérateur de début et un itérateur de fin :

```
find (v.begin (), v.end (), valeur);
search (v.begin (), v.end (), v2.begin (), v2.end ());
```

Ils renvoient un itérateur sur l'emplacement de la valeur trouvée.

Exemple d'utilisation :

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <string.h>

using namespace std;

int main () {
  const char * s = "Les sanglots longs des violons de l'automne";
  const char * s1 = "violon";
  vector<char> v (s, s + strlen (s));
  vector<char> v2 (s1, s1 + strlen (s1));
  vector<char>::iterator it;

  it = find (v.begin (), v.end (), 'v');
  copy(it, v.end(), ostream_iterator<char>(cout, ""));
  cout << endl;
  it = search (v.begin (), v.end (), v2.begin (), v2.end ());
  copy(it, v.end(), ostream_iterator<char>(cout, ""));
  cout << endl;

  return 0;
}
```

7.13 L'algorithme replace

L'algorithme replace remplace une valeur par une autre entre deux itérateurs.

```
replace (v.begin (), v.end (), valeur, autreValeur);
```

Exemple d'utilisation :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main () {
  vector<int> v (20, 10);

  copy(v.begin (), v.end (), ostream_iterator<int>(cout, ", "));
  cout << endl;
  replace (v.begin (), v.end (), 10, 0);
  copy(v.begin (), v.end (), ostream_iterator<int>(cout, ", "));
```

```
  cout << endl;

  return 0;
}
```

7.14 Les algorithmes remove et remove_if

L'algorithme remove ote une valeur entre deux itérateurs.

```
remove (v.begin (), v.end (), valeur);
```

L'algorithme remove_if ote les valeurs pour lesquelles une fonction renvoie true, entre deux itérateurs.

```
remove_if (v.begin (), v.end (), fonction);
```

Exemple d'utilisation :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

bool estPair (int n) { return (n & 1) == 0; }

int main () {
  vector<int> v;

  for (int i = 0; i < 20; i++)
    v.push_back (i);
  copy(v.begin (), v.end (), ostream_iterator<int>(cout, ", "));
  cout << endl;

  vector<int>::iterator fin = remove (v.begin (), v.end (), 15);
  copy(v.begin (), fin, ostream_iterator<int>(cout, ", "));
  cout << endl;

  fin = remove_if (v.begin (), fin, estPair);
  copy(v.begin (), fin, ostream_iterator<int>(cout, ", "));
  cout << endl;

  return 0;
}
```

7.15 L'Algorithme random_shuffle

L'algorithme random_shuffle permet de mélanger aléatoirement une séquence :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main () {
  vector<int> v;

  for (int i = 0; i < 20; i++)
    v.push_back (i);
  copy(v.begin (), v.end (), ostream_iterator<int>(cout, ", "));
  cout << endl;

  random_shuffle (v.begin (), v.end ());
  copy(v.begin (), v.end (), ostream_iterator<int>(cout, ", "));
  cout << endl;

  return 0;
}
```

7.16 L'Algorithme binary_search

L'algorithme binary_search permet de faire une recherche efficace dans une séquence ordonnée :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <stdlib.h>

using namespace std;

int main () {
  vector<int> v;

  for (int i = 0; i < 20; i++)
    v.push_back (i);
  random_shuffle (v.begin (), v.end ());
  sort (v.begin (), v.end ());
  copy(v.begin (), v.end (), ostream_iterator<int>(cout, ", "));
  cout << endl;
```

```
  if (binary_search (v.begin (), v.begin () + 11, 10))
    cout << "10 est present parmi les 11 premiers entiers\n";

  return 0;
}
```

7.17 L'Algorithme for_each

L'algorithme for_each permet d'appliquer une fonction à une ensemble de valeurs :

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void print (int i) { cout << i << " "; }

int main () {
  vector<int> v;

  for (int i = 0; i < 20; i++)
    v.push_back (i);
  for_each (v.begin (), v.end (), print);
  cout << endl;

  return 0;
}
```

7.18 L'Algorithme count_if

L'algorithme count_if permet de compter le nombre d'éléments d'une séquence qui satisfont une fonction :

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool grand (int i) { return i > 10; }

int main () {
  vector<int> v;

  for (int i = 0; i < 20; i++)
    v.push_back (i);
  int n = count_if (v.begin (), v.end (), grand);
```

```
  cout << n << " valeurs sont grandes (>10)\n";
  cout << endl;

  return 0;
}
```