INF 2005 Programmation orientée objet avec C++

Texte 3

Version du 12 janvier 2015

1 Fonctions

Dans les modules précédents, nous avons utilisé une fonction, la fonction `main()`. Elle nous a permis d'exécuter des opérations en utilisant des fonctions prédéfinies : les librairies. Toutefois, dans plusieurs cas, les programmeurs doivent suivre des méthodes de développement de type modulaire, en utilisant des sous-programmes ou de petits programmes autonomes qu'on appelle des fonctions. Cette approche facilite le développement des logiciels à partir d'éléments autonomes connectés par une structure simple et cohérente et augmente ainsi l'extensibilité et la réutilisabilité.

1.1 Définition d'une fonction

Une fonction est définie par son type, un identificateur et ses paramètres. Elle se présente de la manière suivante :

```
Type identificateur (paramètres)
{
      //corps de la fonction
}
```

où le type de la fonction n'est rien d'autre que la valeur de retour de la fonction. L'identificateur est le nom donné à la fonction; ce nom doit être assez significatif pour décrire le plus fidèlement possible la tâche que doit effectuer la fonction. Il y a également les paramètres dont la syntaxe est la suivante :

```
Type identificateur (paramètres)
{
  //Début du corps de la fonction
  type parametre1 [= valeur par défaut],
  type parametre2 [= valeur par défaut],
  ...,
  type parametreN [= valeur par défaut]
}
```

où `type` est le type des paramètres (`int`, `float`, etc.) et valeur par défaut est la valeur qui suit immédiatement le paramètre; il n'est pas obligatoire de donner une valeur par défaut.

Voyons un exemple, soit la déclaration d'une fonction qui permet de faire la somme de deux nombres, puis d'afficher cette valeur. La valeur de retour de la fonction est signifiée par `return`.

```
// exemple de déclaration d'une fonction
int somme (int premierNombre, int deuxiemeNombre)
{
 return (premierNombre + deuxiemeNombre);
 // retourne la somme
}
```

Nous constatons, dans cet exemple, que le type de la fonction est `int`; l'identificateur est `somme`, un nom significatif puisque cette fonction fait la somme de deux nombres; les paramètres sont `premierNombre` et `deuxiemeNombre`.

Notons que si une fonction ne renvoie pas de valeur; on lui donnera le type `void`. De plus, si elle n'attend pas de paramètres dans sa liste de paramètres, sa liste sera `void` ou n'existera pas. Enfin, il n'est pas nécessaire de placer `return` à la fin d'une fonction qui ne retourne pas de valeur. Voyons un deuxième exemple, celui d'une fonction qui ne fait rien, donc qui ne retourne aucune valeur.

```
// une fonction qui ne fait rien
void Rien()
{
      return; // cette ligne est facultative.
}
```

Cette fonction n'a pas de paramètres, c'est pourquoi il n'y rien entre les parenthèses; elle ne retourne aucune valeur d'où la présence de `void` comme type de la fonction et elle ne fait absolument *rien,* comme son identificateur ou son nom l'indique. Le `return` entre l'accolade ouvrante et l'accolade fermante est facultatif.

1.2 L'appel de fonctions

Une fois les fonctions écrites sous la forme de sous-programmes, il faut pouvoir les réunir pour créer un logiciel. Pour cela, le programmeur ou développeur de logiciel en C*++* appelle chacune de ces fonctions pour former une structure simple et cohérente : le programme ou le logiciel. L'appel des fonctions s'effectue comme l'illustre l'exemple suivant :

```
// Exemple d'appel de fonctions
void Rien ();
// Appel de la fonction « Rien »; voir le deuxième exemple
int sommecalculer = somme (2,3);
// Voir l'exemple de la fonction somme
```

Notons que dans l'appel d'une fonction, les parenthèses sont obligatoires, même si la fonction n'a pas de paramètres.

1.3 La déclaration de fonctions

Toute fonction doit être déclarée avant d'être appelée pour la première fois; la définition d'une fonction peut faire office de déclaration de cette fonction. La déclaration d'une fonction est très importante, car il se peut que la fonction soit appelée dans une autre fonction définie avant elle. Dans le langage de programmation C++, il est important de déclarer une fonction pour éviter des erreurs lors de la compilation.

Le rôle des déclarations de fonctions est de signaler leur existence aux compilateurs afin qu'ils puissent les utiliser en se rapportant à la définition de ces fonctions dans le programme ou bien dans un autre fichier.

Comme nous l'avons déjà vu, la syntaxe d'une déclaration prend la forme suivante :

```
Type identificateur (les paramètres)
```

Le *type* représente le type de la valeur retournée par la fonction, `identificateur` est le nom donné à cette fonction et les paramètres ne sont que la liste des types des paramètres séparés par une virgule. Voici un exemple de déclaration d'appel et de définition de la fonction `somme`.

```
// Exemple de déclaration de fonctions
int somme (int, int); // Déclaration de la fonction somme
// Fonction principale qui fait appel à la fonction somme
# include <iostream> using namespace std;
int main (void)
{
 int sommecalculer = somme (2,5);
 cout<<" sommecalculer = : "<< sommecalculer << endl;
 return (0);
 // Définition de la fonction somme
 int somme (int premierNombre, int deuxiemeNombre)
 {
      return (premierNombre + deuxiemeNombre); // Retourne la
                                               // somme
}
```

En langage de programmation C++, il est possible de donner des valeurs par défaut aux paramètres dans la déclaration et ces valeurs peuvent être différentes de celles que l'on

peut trouver dans une autre déclaration. Dans ce cas, les valeurs utilisées sont celles de la déclaration visible lors de l'appel de la fonction.

1.4 La surcharge des fonctions

En langage de programmation C++, plusieurs fonctions peuvent porter le même nom, c'est-à-dire avoir le même identificateur. Ce qui distingue ces fonctions l'une de l'autre, c'est la liste de leurs paramètres; cette liste constitue la signature des fonctions. Lorsque des fonctions portent le même nom et qu'elles ne se différencient que par leur signature, elles sont dites fonctions *surchargées*. Lors de l'appel d'une fonction dont le nom est commun à plusieurs, la fonction qui sera choisie sera celle dont les paramètres passés lors de l'appel se rapprochent le plus de la signature de la fonction.

```
//Exemple d'utilisation de fonctions surchargées
// Première définition de la fonction.
int somme (int premierNombre, int deuxiemeNombre)
{
  return (float) premierNombre + deuxiemeNombre;
}
// Deuxième définition de la fonction.
int somme (float premierNombre, float deuxiemeNombre)
{
  return (premierNombre /deuxiemeNombre);
}
```

Ces deux fonctions portent le même nom et n'ont pas les mêmes signatures; elles seront donc toutes les deux acceptées par le compilateur. Si nous demandons l'opération `somme (4, 5)`, la première fonction sera appelée; par contre, si nous demandons l'opération `somme (1.20, 3.12)`, c'est la deuxième fonction qui sera appelée. Par ailleurs, si nous demandons `somme (4.5, 9)`, la valeur 4.5 sera convertie en `entier` et la première fonction sera utilisée, selon les règles de priorité définies par le compilateur.

Il faut éviter de surcharger les fonctions dont les paramètres ont des valeurs par défaut, car le compilateur ne pourra pas faire la distinction entre ces fonctions.

1.5 La fonction *inline*

En programmation C++, le mot clé `inline` placé devant le type d'une fonction propose au compilateur de ne pas instancier cette fonction. Cela veut dire que le compilateur remplace l'appel d'une fonction par le code correspondant. Cette possibilité offerte en programmation C++ a des avantages et des inconvénients.

Quand la taille de la fonction est importante ou que cette fonction est appelée plusieurs fois dans un programme informatique, la taille de ce dernier augmente puisque, à chaque appel de la fonction, le code correspondant est écrit. En revanche, le programme devient nettement plus rapide, car tout le processus d'appel de fonction, de passage des paramètres et des valeurs de retour est évité. En pratique, cette utilisation est réservée pour les fonctions dont la taille est petite. Il faut cependant faire attention à l'utilisation du `inline`, car le compilateur peut décider d'agir selon son propre processus d'optimisation.

Il existe également des restrictions à utiliser cette fonction, car elle ne peut pas être une fonction récursive. Comme elle ne sera pas instanciée, on ne pourra donc pas utiliser des pointeurs pour une telle fonction. Voyons un exemple d'une fonction `inline`.

```
// Exemple d'une fonction inline
inline int somme (int premierNombre, int deuxiemeNombre)
{
return (premierNombre + deuxiemeNombre)
}
```

Notons que, pour le type de fonction de l'exemple, l'utilisation de mot clé `inline` devant le type de la fonction est tout à fait justifiée.

1.6 Les fonctions statiques

En langage de programmation C++, des fonctions peuvent être définies comme externes au programme principal; ces fonctions dites externes sont regroupées dans un fichier et le fichier est déclaré dans le programme principal. Il est aussi intéressant de définir des fonctions locales propres à un fichier, soit parce que la fonction n'a d'intérêt que pour ce fichier, soit pour éviter des conflits de nom et de signature de la fonction.

À cette fin, le langage de programmation C++ fournit le mot clé `static`. Une fois placé devant la définition et éventuellement la déclaration de la fonction, il la rend unique et utilisable seulement dans ce fichier. Les fonctions dites statiques se comportent exactement comme les fonctions classiques.

Voyons un exemple de déclaration et de définition d'une fonction statique.

```
//Exemple de fonction statique
// Déclaration d'une fonction statique
static int somme (int premierNombre, int deuxiemeNombre)
static int somme (int, int)
// Définition d'une fonction statique
```

```
{
  return (premierNombre + deuxiemeNombre)
}
```

1.7 Le passage des valeurs par référence

Jusqu'à présent, dans les appels de fonctions, nous avons passé les paramètres à la fonction appelée en utilisant une valeur. Ceci signifie que lorsque l'argument (ou le paramètre) est transmis par une valeur à une fonction, c'est une copie de cette valeur qui est transmise; toutes les modifications effectuées sur la valeur sont faites sur la copie, donc ne modifient aucunement la valeur originale.

Lorsqu'il faut faire passer un nombre important de valeurs, cette façon de faire ralentit le programme. Pour y remédier, le langage de programmation C++ propose de passer les arguments par référence, ce qui signifie que les modifications effectuées sur ces paramètres sont faites directement sur les valeurs originales. Cette méthode est surtout utilisée pour sa performance, mais peut être dangereuse dans certains cas. La déclaration ou la définition des paramètres s'effectue de la manière suivante :

```
Type du paramètre & paramètre.
```

Par exemple :

```
int & premierNombre.
```

Voyons un programme où le passage des valeurs se fait par référence :

```
// Exemple de passage de valeurs par référence.
// Déclaration de la fonction
#include <iostream>
using namespace std;
int somme (int &);
// Fonction principale
int main ()
{
  int un = 4, deux= 3;
  cout<< "La valeur de un avant l'appel de la fonction est de :
  un= "<<un<<endl;
  int sommecalculer = somme (un);
  cout<< "La valeur de un après l'appel de la fonction : un=
  "<<un<< " "<<endl;
  cout<< "sommecalculer : sommecalculer = "<<
  sommecalculer<<endl;
  return 0;
}
```

```
// Définition de la fonction somme
int somme (int & premierNombre)
{
  premierNombre += premierNombre;
  return (premierNombre);
      }
```

Les résultats affichés sont :

```
// Résultats affichés
La valeur de un avant l'appel de la fonction est de : un= 4
La valeur de un après l'appel de la fonction : un= 8
sommecalculer : sommecalculer = 8
```

2. Tableaux

Une des structures de données les plus importantes en informatique est le tableau. Un tableau en informatique est considéré comme un emplacement mémoire (ou une zone mémoire) composé de plusieurs cases consécutives permettant de manipuler des données du même type. Cette structure permet de faire certaines opérations sur des données dont la gestion, le tri et la recherche d'un élément placé dans une liste, l'enregistrement des données dans un tableau. Nous allons voir comment déclarer et initialiser un tableau et voir quelques exemples liés à la manipulation des tableaux.

2.1 La déclaration et l'initialisation d'un tableau

Comme le tableau permet de représenter un ensemble de données de même type, on repère ces données par des indices représentant les emplacements mémoires consécutifs. En programmation C++, comme toutes les variables, les tableaux doivent être déclarés.

Un tableau représente une matrice et la syntaxe de sa déclaration comprend :

```
type identificateur [le nombre d'emplacements mémoires]
```

où le type est le type des données que doit contenir le tableau et identificateur le nom donné à ce tableau. Étant donné qu'un tableau représente une matrice alors, d'une manière générale, il peut avoir plusieurs lignes et plusieurs colonnes, donc plusieurs dimensions. Voici la déclaration générale d'un tableau.

```
Type identification [Emplacement Mémoire1][ Emplacement
Mémoire2]... ... [Emplacement MémoireN]
```

Voyons un exemple de déclaration de trois tableaux.

```
Tint Tableaudesentiers [10];
char Tableaudeschainesdecaractères [10];
int Nombre [20][10];
```

Quant à l'initialisation, elle peut se faire lors des déclarations ou à l'intérieur d'un bloc d'instructions d'une fonction. Voyons deux exemples des possibilités d'initialisation.

```
// Exemple d'initialisation de tableaux lors de la déclaration
de la fonction
int Tableaudesentiers[10] = {0,1,2,3,4,5,6,7,8,9};
int Nombre[3][2] ={{1,2}, {12,9},{5,4}};
```

```
// Exemple de déclaration et d'initialisation à l'intérieur //
d'un bloc d'instructions (fonction)
//#include <iostream.h> si vous utilisez windows
//#include <iomanip.h> si vous utilisez windows
#include <iostream>
#include <iomanip>
using namespace std;
int main ()
{
  const int dimensiontableau =5;
  int TableauEntier [dimensiontableau];
  for (int i=0; i < dimensiontableau;i++)
  {
      TableauEntier[i]=i+1;
  }
  for (int j=0; j < dimensiontableau;j++)
  {
      cout << " TableauEntier [" << j <<"] = ";
      cout << TableauEntier[j] <<endl;
  }
return 0;
}
```

Les résultats qu'affichera cette fonction sont :

```
TableauEntier [0] = 1
TableauEntier [1] = 2
TableauEntier [2] = 3
TableauEntier [3] = 4
TableauEntier [4] = 5
```

Notons qu'un tableau peut être défini sans spécification de sa taille; dans ce cas, sa taille est celle des données qu'on lui fournit lors de son initialisation. La taille du tableau est de 4.

2.2 Le passage de tableau en paramètre

Comme les variables, un tableau peut aussi être le paramètre d'une fonction. Ce paramètre, doit toujours être passé par référence. Par exemple, si un tableau des notes obtenues dans le cours INF 2005 est défini dans un tableau dont la taille est de 20 étudiants, alors la déclaration sera la suivante :

```
int NotesCoursinf2005 [20];
```

Si, pour une raison quelconque, le chargé de cours décide d'ajouter deux points aux notes de tous les étudiants, alors il pourra définir une fonction `AjouterDeux`, qui aura comme paramètre le tableau `NotesCoursinf2005` et un entier `int Deux=2`.

L'appel de cette fonction sera fait comme suit :

```
AjouterDeux (NotesCoursinf2005,Deux);
```

2.3 La manipulation des tableaux à plusieurs dimensions

La manipulation des tableaux à plusieurs dimensions se fait en manipulant des lignes et des colonnes. Un tableau se définit comme une représentation matricielle des données; il est donc formé d'un ensemble de lignes et de colonnes. La définition d'une fonction à deux dimensions se fait de la manière suivante :

```
int TableauNotes [3][4]
```

Dans le tableau suivant à deux dimensions, nous avons représenté tous les exercices effectués dans le cadre du cours INF 2005 par tous les étudiants. Chaque ligne représente un étudiant et chaque colonne correspond à chaque exercice.

La représentation de ce tableau se fait de la manière suivante :

```
TableauNotes[0][0] TableauNotes[3][1]  TableauNotes[3][2]  TableauNotes[3][3]

TableauNotes[1][0] TableauNotes[3][1]  TableauNotes[3][2]  TableauNotes[3][3]

TableauNotes[2][0] TableauNotes[3][1]  TableauNotes[3][2]  TableauNotes[3][4]
```

3. Techniques de tri et méthodes de recherche d'éléments dans les tableaux

Un des algorithmes importants en informatique est le tri et la recherche de données. Les tableaux sont le plus souvent utilisés pour implémenter ces algorithmes.

3.1 Méthode de tri : le tri à bulle

Le tri à bulle est la méthode qui consiste à faire « remonter » de manière graduelle la donnée la plus petite et de faire reculer la plus grande, jusqu'au classement complet, par ordre croissant. Ces données sont les éléments d'un tableau. Le terme « bulle » vient, par analogie, du fait que les éléments triés pendant l'opération sont des bulles d'air qui remontent à la surface d'un liquide.

L'algorithme général du tri à bulle est le suivant :

```
// Tri à bulle
void Tri_bulle (int a[], int n)
{
  int i, j; // Index des éléments à comparer
  int tmp; // Variable tampon
  // Comparaison des éléments adjacents
  for (i=0; i< n-1; i++)
      for(j=n-1 ;j>=i; j--)
       {
          if(a[j-1]>a[j])
           {
            tmp = a[j-1];
            a[j-1]=a[j];
            a[j]=tmp;
            }
       }
  }
}
```

Exemple :

```
/// Exemple de tri a bulle
#include <iostream>
#include <iomanip>
using namespace std; // pour linux
int main ()
{
  const int NombreElement = 10;
```

```
 int Tampon = 0;
 int TableauNote[10]={1,5,7,89,64,75,32,45,56,10};
 cout<< "Ordre initial des éléments du tableau est : "<<endl;
 for(int i=0;i< NombreElement; i++)
 {
   cout<< TableauNote[i]<<endl;
 }
 for(int iteration =0;iteration < NombreElement-1;iteration++)
      for(int i=0; i< NombreElement-1; i++)
            if(TableauNote[i]> TableauNote[i+1])
             {
                  Tampon = TableauNote[i];
                  TableauNote[i]= TableauNote[i+1];
                  TableauNote [i+1]= Tampon;
              }
cout<<endl<<"Les données dans l'ordre croissant sont: "<<endl;
for(int j=0; j< NombreElement-1; j++)
  cout<<" "<<TableauNote[j]<<" "<<endl;
return 0;
}
```

3.2 Les méthodes de recherche

La recherche constitue le processus par lequel on retrouve un élément spécifique dans un tableau. Voyons deux méthodes de recherche souvent utilisées par les informaticiens : la *recherche linéaire* et la *recherche binaire*.

La recherche linéaire consiste à définir une clé et à la comparer à chaque élément du tableau. Les éléments de ce tableau étant dans un ordre quelconque, l'élément recherché peut aussi bien être le premier que le dernier. Si cet élément appartient à l'une des premières cases du tableau, le nombre de comparaisons pour l'atteindre sera faible. Dans le cas contraire, il peut être très élevé. En moyenne, pour un tableau de *n* éléments, le programme effectuera n/2 comparaisons.

Toutefois, cette méthode fonctionne bien seulement dans le cas des tableaux de taille restreinte ou lorsque les éléments ne sont pas triés. Dans le cas où les éléments sont déjà triés, on utilise une méthode de recherche plus efficace : la recherche binaire. L'exemple qui suit illustre l'implémentation de la recherche linéaire.

```
// Recherche linéaire dans un tableau
#include <iostream>
using namespace std;
int Recherche_lin(int [], int, int);
main()
{
```

```
 const int Dim_Tableau = 100;
 int a[Dim_Tableau], CleRecherche, element;
 for (int x = 0; x < Dim_Tableau; x++) // Génération des données
      a[x] = 2 * x;
 cout << "Entrer la clé de recherche :" << endl;
 cin >> CleRecherche;
 element = Recherche_lin(a, CleRecherche, Dim_Tableau);
 if (element != -1)
      cout << " Position de la valeur trouvée " << element <<
endl;
 else
      cout << "Valeur introuvable" << endl;
      return 0;
 }
int Recherche_lin(int table[], int Cle, int Dimension)
{
   for (int n = 0; n < Dimension; n++)
      if (table[n] == Cle)
            return n;
    return 0;
}
```

Quant à l'algorithme de recherche binaire, il élimine à chaque comparaison la moitié des éléments du tableau à chercher. Il considère l'élément du milieu (valeur médiane) du tableau et le compare à la clé de recherche. Si ces deux éléments ont la même valeur, l'élément recherché est trouvé. Dans le cas contraire, le problème se résume à trouver la valeur recherchée dans l'une des deux parties du tableau : si la clé de recherche est plus faible que la valeur médiane du tableau, la recherche s'effectue dans la première moitié; dans le cas contraire, elle s'exécute dans la seconde moitié.

Ce processus se poursuit jusqu'à ce que la clé de la recherche corresponde à la valeur médiane d'un sous-tableau ou que le dernier sous-tableau se réduise à un élément qui ne correspond pas à la clé de recherche. Dans ce dernier cas, l'élément recherché n'existe pas. Le programme qui suit illustre une implémentation itérative de la recherche binaire.

La fonction `RechercheBin` reçoit quatre arguments :

- un tableau d'entiers `b`;
- un entier `CleRecherche`;
- les variables `inferieur` et `superieur` qui représentent respectivement le premier et le dernier élément du tableau.

Si la clé de recherche ne correspond pas à la valeur médiane, les variables `inferieur` et `superieur` sont rajustées de façon à restreindre la recherche dans le tableau. Autrement dit, si `CleRecherche` est plus faible que la valeur médiane du tableau, la variable `superieur` est ajustée à `milieu – 1` et la recherche s'effectue dans la première moitié du tableau (de `inferieur` à `milieu – 1`).

Dans le cas contraire, elle s'exécute dans la seconde moitié (`de milieu + 1` à `superieur`). Pour un tableau de `n` éléments, le nombre de comparaisons à effectuer est de l'ordre de `[(ln n)/(ln 2)]`.

Dans notre exemple, on manipule un tableau de 15 éléments, seulement quatre comparaisons sont nécessaires pour trouver l'élément recherché.

```
// Recherche binaire dans un tableau
// #include <iostream.h> // à utiliser si vous êtes sur Wind
// #include <iomanip.h> // à utiliser si vous êtes sur wind
#include <iostream>
#include <iomanip>
using namespace std;
int Recherche_bin(int [], int, int, int, int);
void Imprime_entete(int);
void Imprime_ligne(int [], int, int, int, int);
main()
{
  const int Dim_Tableau = 15;
  int a[Dim_Tableau], Cle, resultat;
  for (int i = 0; i < Dim_Tableau; i++)
            a[i] = 2 * i; // Génération de données
      cout << "Entrer un nombre entre 0 et 28: ";
cin >> Cle;
Imprime_entete(Dim_Tableau);
resultat = Recherche_bin(a, Cle, 0, Dim_Tableau - 1,
Dim_Tableau);
if (resultat != -1)
            cout << endl << Cle << " est retrouvé en position
"<< resultat << endl;
      else
cout << endl << Cle << " introuvable" << endl;
return 0;
}
// Recherche binaire
int Recherche_bin(int b[], int CleRecherche, int inferieur,
int superieur, int grandeur)
{
      int milieu;
while (inferieur <= superieur) {
```

```
            milieu = (inferieur + superieur) / 2;
Imprime_ligne(b, inferieur, milieu, superieur, grandeur);
if (CleRecherche == b[milieu]) // Élément recherché est trouvé
                  return milieu;
            else if (CleRecherche < b[milieu])
                  superieur = milieu - 1;
// Recherche dans la partie inférieure
            else
                  inferieur = milieu + 1;
// Recherche dans la partie supérieure
      }
return -1; // Clé de recherche introuvable
}

// Imprimer en-tête à l'écran
void Imprime_entete(int grandeur)
{
      cout << endl << "Positions:" << endl;
for (int i = 0; i < grandeur; i++)
            cout << setw(3) << i << ' ';
// setw(3) Limite le nombre de caractères à 3
      cout << endl;
for (i = 1; i <= 4 * grandeur; i++)
            cout << '-';
      cout << endl;
}

// Impression ligne par ligne des éléments du tableau à être //
traités
void Imprime_ligne (int b[], int inferieur, int mid, int
superieur, int grandeur)
{
      for (int i = 0; i < grandeur; i++)
            if (i < inferieur || i > superieur)
                  cout << " ";
            else if (i == mid)
                  cout << setw(3) << b[i] << '*'; // Étiqueter
la valeur du milieu
            else
                  cout << setw(3) << b[i] << ' ';
      cout << endl;
}
```

4. Manipulation des chaînes de caractères

En langage C++, comme dans beaucoup de langage de programmation, on assimile une chaîne de caractères à un tableau de plusieurs éléments de type `char`, qui se termine par un caractère nul. Par conséquent, on parcourt une chaîne de caractères comme si on parcourrait un tableau.

Une chaîne composée de *n* éléments sera en fait un tableau de *n+1* éléments de type `char`.

4.1 La déclaration d'une chaîne de caractères

Pour déclarer une chaîne de caractères, il suffit de déclarer un tableau de caractères. La syntaxe de cette déclaration est la suivante :

```
char Nom_du_tableau[Nombre_d_elements]
```

Voici un exemple :

```
char Chaîne []={ 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

Ici, `Chaîne` est déclarée comme un tableau de caractères qui est initialisé avec plusieurs caractères. Le dernier caractère `\0'` est un caractère nul et qui représente la fin de la chaîne.

Mais bien que cette approche par caractère fonctionne, elle est difficile à saisir et bien trop sujette aux erreurs de frappe. Par conséquent, il est aussi simplement possible de déclarer cette chaîne de la manière suivante :

```
char Chaîne = "Bonjour";
```

La longueur d'une chaîne est le nombre de caractères, y compris le caractère nul. Vous pouvez aussi créer des tableaux de caractères non initialisés. Comme pour tout autre tableau, vous devez vérifier que vous n'en mettez pas plus qu'il y a d'éléments réservés.

```
include <iostream>

int main()
{
  char buffer[80];
  std::cout << "Entrez la chaîne : ";
  std::cin >> buffer;
  std::cout << "Contenu du tampon : " << buffer <<
```

```
  std::endl;
  return 0;
 }
```

En compilant ce programme, on constate bien que le résultat serait simplement le mot `Bonjour`. Ce programme contient deux erreurs potentielles :

- la première est que si l'utilisateur tape plus de 79 caractères, `cin` écrira après la fin du tampon;
- la seconde est qu'un caractère d'espacement est considéré par `cin` comme une fin de chaîne et qu'il arrête d'écrire dans le tampon, d'où le résultat `Bonjour`.

Le programme peut être corrigé en appelant la méthode `get()` sur `cin`. Ainsi, nous aurons :

```
include <iostream>
using namespace std;

int main()
{
   char buffer[80];
   cout << "Entrez une chaine : ";
   cin.get(buffer, 79);    // Lire 79 caractères ou
                          //jusqu'à newline

   cout << "Contenu du tampon : " << buffer << endl;
   return 0;
}
```

Le résultat de l'exécution est :

```
Bonjour mes amis.
```

4.2 Les fonctions de manipulation des chaînes de caractères

La bibliothèque de C++ fournit plusieurs fonctions qui permettent de traiter les chaînes de caractères. C++ hérite de C les fonctions de manipulation des chaînes de type C. Ces fonctions se trouvent dans le fichier d'en-tête *<string.h>*. II faut ajouter la ligne suivante en début de programme pour manipuler les chaînes de caractères :

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

Notez que dans la version actuelle de C++ vous pouvez omettre l'utilisation du `.h`.

La bibliothèque `<string.h>` contient de nombreuses fonctions qui permettent de simplifier l'utilisation et la manipulation de chaînes de caractères. Parmi elles, les fonctions `strcpy()` et `strncpy()` permettent de copier une chaîne dans une autre.

- `strcpy()` copie le contenu entier d'une chaîne dans un tampon, tandis que `strncpy()` ne copie qu'un nombre de caractères;
- `strlen` retourne la longueur d'une chaîne de caractères;
- `strcat` permet de concaténer une chaîne à une autre;
- `strcmp` permet de faire la comparaison lexicographique de deux chaînes de caractères.
- etc.

```
include <string.h>
#include <iostream>
using namespace std;
int main()
{
  char Chaine1[] = "Nul n'est prophète en son pays";
  char Chaine2[80];
  strcpy(Chaine2,Chaine1);
  cout << "Chaine1 : " << Chaine1 << endl;
  cout << "Chaine2 : " << Chaine2 << endl;
  return 0;
 }
```

Cet exemple simple permet de déclarer deux tableaux de chaîne. Le programme copie les données d'une chaîne dans une autre.

- La fonction `strcpy()`, présente un inconvénient puisqu'elle peut copier les caractères au-delà du dernier élément du tableau cible si le tableau source est plus grand.
- L'utilisation de la fonction `strncpy()` permet de remédier à ce problème, car elle permet de limiter le nombre de caractères copiés. Elle effectue la copie dans le tampon de destination jusqu'à ce qu'elle rencontre un caractère nul ou qu'elle atteigne le nombre maximal de caractères dans le tampon source.

```
include <string.h>
#include <iostream>
using namespace std;
```

```
int main()
{
  const int LongMax = 80;
  char Chaine1[] = "Nul n'est prophète en son pays";
  char Chaine2[LongMax+1];

  strncpy(Chaine2,Chaine1,LongMax);

  cout << "Chaine1 : " << Chaine1 << endl;
  cout << "Chaine2 : " << Chaine2 << endl;
  return 0;
 }
```

4.3 Recherche de chaîne de caractères : **`find()`**

C++ possède des fonctions qui s'appliquent à une instance d'un objet. Ce type de fonction s'appelle « fonction d'instance ». C'est le cas notamment de la fonction `find()`.

La fonction `find()` permet de chercher la position d'une chaîne de caractères. Elle prend comme argument la chaîne de caractères à chercher et renvoie la position de la première occurrence de cette chaîne. Si la chaîne n'est pas trouvée alors la fonction renvoie `-1`.

```
#include <string.h>
#include <iostream>
using namespace std;

int main()
{
  string a = "Aujourdhui";
  int p = a.find("hui");
  cout << p << endl;
}
```

Le résultat est `7`, ce qui correspond bien à la première occurrence de la chaîne `hui` dans la chaîne `Aujourdhui`.

Il est possible aussi de rajouter un deuxième argument à la fonction `find()` pour choisir une position à laquelle la recherche de la chaîne doit commencer.

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

```
using namespace std;
int main()
{
   string a = "Aujourd'hui nous sommes lundi";
   int i = a.find('u');

   while(i != -1)
   {
       cout << i << endl;
       i = a.find('u', i+1);
   }
}
```

Dans cet exemple, le programme recherche la première occurrence du caractère `u` dans la chaîne `Aujourd'hui nous sommes lundi`. Lorsqu'une occurrence est trouvée alors il l'affiche et l'algorithme se poursuit jusqu'à la prochaine occurrence.

Notez aussi que la fonction `rfind()` permet de faire une recherche de chaîne de caractères dans une autre chaîne de caractères, à l'exception qu'ici la recherche s'effectue de la fin vers le début.