Chapitre IV : Tableaux, Pointeurs et Chaines de caractères en C++

IV.1 Tableaux unidimensionnels

Une variable entière de type ***int*** ne peut contenir qu'une seule valeur. Si on veut stocker en mémoire un ensemble de valeurs, il faut utiliser une structure de données appelée tableau qui consiste à réserver espace de mémoire contiguë dans lequel les éléments du tableau peuvent être rangés. On peut distinguer deux sortes de tableaux unidimensionnels :

- Les tableaux statiques : Ceux dont la taille est connue à l'avance, et
- Les tableaux dynamiques : ceux dont la taille peut varier en permanence.

IV.1.1 Tableaux unidimensionnels statiques

IV.1.1.1 Déclaration et initialisation

Comme toujours en C++, une variable est composée d'un nom et d'un type. Comme les tableaux sont des variables, cette règle reste valable. Il faut juste ajouter une propriété supplémentaire, la taille du tableau.

Syntaxe :

```
type nomDuTableau [taille];
```

- ***type***: définit le type des éléments que contient le tableau.
- ***nomDuTableau***: le nom que l'on décide de donner au tableau.
- ***taille***: nombre entier qui détermine le nombre de cases que le tableau doit comporter.

- Rappelez-vous qu'un tableau en langage C++ est composé uniquement d'éléments de même type.
- Le nom du tableau suit les mêmes règles qu'un nom de variable.

Exemple: [4c]

```
int meilleurScore [5];
char voyelles [6];
double notes [20];
```

Il est possible aussi d'initialiser le tableau à la déclaration en plaçant entre accolades les valeurs, séparées par des virgules.

Exemple :

```
int meilleurScore [5] = {100, 432, 873, 645, 314};
```

- Le nombre de valeurs entre accolades ne doit pas être supérieur au nombre d'éléments du tableau.
- Les valeurs entre accolades doivent être des constantes, l'utilisation de variables provoquera une erreur du compilateur.
- Si le nombre de valeurs entre accolades est inférieur au nombre d'éléments du tableau, les derniers éléments sont initialisés à 0.
- Si le nombre de valeur entre accolades est nul, alors tous les éléments du tableau s'initialisent à zéro.

Exemple :

```
int meilleurScore [5] = {};
```

IV.1.1.2 Manipulation des éléments

Un élément du tableau (repéré par le nom du tableau et son indice) peut être manipulé exactement comme une variable, on peut donc effectuer des opérations avec (ou sur) des éléments de tableau.

Le point fort des tableaux, c'est qu'on peut les parcourir en utilisant une boucle. On peut ainsi effectuer une action sur chacune des cases d'un tableau, l'une après l'autre : par exemple afficher le contenu des cases.

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

int main()
{
  int const taille(10); //taille du tableau
 int tableau[taille]; //declaration du tableau

 for (int i(0); i<taille; i++ )
 {  tableau[i]=i*i;     cout<<"Le tableau ["<<i<<"] contient la valeur "<<tableau[i]<<endl;  }
 return 0;
}
```

Affichage de l'exécution:

```
Le tableau [0] contient la valeur 0
Le tableau [1] contient la valeur 1
Le tableau [2] contient la valeur 4
Le tableau [3] contient la valeur 9
Le tableau [4] contient la valeur 16
Le tableau [5] contient la valeur 25
Le tableau [6] contient la valeur 36
Le tableau [7] contient la valeur 49
Le tableau [8] contient la valeur 64
Le tableau [9] contient la valeur 81

Process returned 0 (0x0)   execution time : 1.936 s
Press any key to continue.
```

IV.1.1.3 Tableaux et fonctions

Au même titre que les autres types de variables, on peut passer un tableau comme argument d'une fonction. Voici donc une fonction qui reçoit un tableau en argument :

```
void afficheTableau(int tableau[], int n)
{
 for(int i = 0; i < n; i++)
  cout << tableau[i] << endl;
}
```

À l'intérieur de la fonction *afficheTableau()*, il n'y a aucun moyen de connaître la taille du tableau ! C'est pour cela nous avons ajouté un deuxième argument ***n*** contenant la taille du tableau.

IV.1. 2 Tableaux unidimensionnels dynamiques (classe vector)

Les tableaux que nous avons vus jusqu'ici sont des tableaux statiques. Cette forme de tableaux vient du langage C, et est encore très utilisée. Cependant, elle n'est pas très pratique. En particulier :

- Un tableau de cette forme ne connaît pas sa taille.
- On ne peut pas faire d'affectation globale.
- une fonction ne peut pas retourner de tableaux.

Pour remédier ces trois problèmes, Le C++ a introduit la notion des ***tableaux dynamiques*** ces derniers sont des tableaux dont le nombre de cases peut varier au cours de l'exécution du programme. Ils permettent d'ajuster la taille du tableau au besoin du programmeur.

IV.1. 2.1 Déclaration

La première différence se situe au début de votre programme. Il faut ajouter la ligne **#include** <**vector**> pour utiliser ces tableaux. Et la deuxième différence se situe dans la manière de déclarer un tableau.

Syntaxe:

```
vector <type> nomDuTableau(taille);
```

Par exemple, pour un tableau de 3 entiers, on écrit :

```
vector<int> tab(3);
```

Pour initialiser les éléments de tab2 aux mêmes valeurs que tab1.

```
vector<int> tab2(tab1);
```

On peut même déclarer un tableau sans cases en ne mettant pas de parenthèses du tout:

```
vector<int> tab;
```

IV.1.2.2 Accès aux éléments

On peut accéder aux éléments de tab de la même façon qu'on accéderait aux éléments d'un tableau statique : **tab[0] = 7**.

Exemple: [4c]

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

int main()
{
  int const TailleTab(5);
//Déclaration du tableau
vector<int> Scores(TailleTab);

//Remplissage du tableau
Scores[0] = 100432; Scores[1] = 87347; Scores [2] = 64523; Scores[3] = 31415; Scores[4] = 118218;

for(int i(0); i<TailleTab; ++i)
   cout << "Meilleur score de joueur " << i+1 << " est : " << Scores[i] << endl;
return 0;

}
```

IV.1.2.2 Modification de la taille

On peut modifier la taille d'un tableau soit en ajoutant des cases à la fin d'un tableau ou en supprimant la dernière case d'un tableau.

Commençons par ajouter des cases à la fin d'un tableau.

a. Fonction push_back()

Il faut utiliser la fonction *push_back()*. On écrit le nom du tableau, suivi d'un point et du mot push_back avec, entre parenthèses, la valeur qui va remplir la nouvelle case.

Exemple :

```
vector<int> tableau(3,2);  //Un tableau de 3 entiers valant tous 2
tableau.push_back(8);  //On ajoute une 4ème case au tableau qui contient la valeur 8
```

b. Fonction pop_back()

On peut supprimer la dernière case d'un tableau en utilisant la fonction ***pop_back()*** de la même manière que push_back(), sauf qu'il n'y a rien à mettre entre les parenthèses.

Exemple :

```
vector<int> tableau(3,2);  //Un tableau de 3 entiers valant tous 2
tableau.pop_back(8);  // Il y a plus que 2 éléments dans le tableau
```

IV.1.2.3 Autres opérations de la classe "vector"

Opération	Description
Accès aux éléments, itérateurs	
front()	retourne une référence sur le premier élément **ex** : vf.**front**() += 11; // +11 au premier élément
back()	retourne une référence sur le dernier élément **ex :** vf.**back**()+=22; // +22 au dernier élément
at()	méthode d'accès avec contrôle de l'existence de l'élément (possibilité de récupérer une erreur en cas de débordement) **ex : for**(**int** i=0; i<vi.**size**(); i++) vi.**at**(i)=rand()%100;
data()	retourne un pointeur sur le premier élément du tableau interne au conteneur **ex: int***p2=vi.**data**(); **for**(**int** i=0; i<vi.**size**(); i++, p2++) **cout**<<*p2<<"-"; **cout**<<endl;
Affectation, Insertion et Suppression d'éléments n'importe où dans le tableau	

assign()	remplace le contenu d'un vecteur par un nouveau contenu en adaptant sa taille si besoin
	ex:
	vector<int> v1; **vector**<int> v2;
	v1.**assign**(10, 50); // v1 remplacé par 10 entiers à 50
	int tab[] = { 10, 20, 30 }; // v2 remplacé par les éléments du tableau tab
	v2.**assign**(tab, tab + 3);
	// affichage des tailles des vecteurs
	cout << **int**(v1.**size**()) << **endl**; cout << **int**(v2.**size**()) << **endl;**

insert(p, x)	ajoute un élément **x** avant l'élément désigné par l'itérateur **p** **ex:** **vector<float>** v; v.**insert**(v.**begin**(),1.5); // ajoute 1.5 avant, au début
insert(p, n, x)	ajoute n copies de **x** avant l'élément désigné par l'itérateur **p** **ex:** v.**insert**(v.**end**(),5,20) ; // ajoute cinq éléments initialisés 20 à la fin

emplace(p,x)	ajoute un élément x à la position désignée par l'itérateur p et décale le reste. Les arguments passés pour x correspondent à des arguments pour le constructeur de x
	ex : p=v.**begin**()+2; // ajoute 100 à la position 2
	v.**emplace**(p, 100);

erase(p)	supprime l'élément pointé par l'itérateur **p** **ex:** //supprime les éléments compris entre les itérateurs premier et dernier **vector<float>**::iterator prem = v.**begin**()+1; **vector<float>**::iterator dern = v.**end()**-1; v.**erase**(prem,dern); affiche_vector(v);
clear()	efface tous les éléments d'un conteneur. Équivalent à c.**erase**(c.begin(), c.end()) **ex:** v.**clear**(); // efface tout le conteneur
Taille et capacité	
size()	retourne le nombre d'éléments du « vector » **ex :** **vector<int>** v(5); **cout**<<v.**size**()<<endl; // 5

resize(nb)
resize(nb, val)
	redimensionne un « vector » avec nb éléments. Si le conteneur existe avec une taille plus petite, les éléments conservés restent inchangés et les éléments supprimés sont perdus. Avec une taille plus grande, les éléments ajoutés sont initialisés avec une valeur par défaut ou avec une valeur spécifiée en val
	ex: **vector<int> v**(5);
	for(unsigned i=0; i<v.**size**(); i++)
	v[i]=i;
	affiche_vector(v); // 0,1,2,3,4
	v.**resize**(7);
	affiche_vector(v); // 0,1,2,3,4,0,0
	v.**resize**(10,99);
	affiche_vector(v); // 0,1,2,3,4,0,0,99,99,99
	v.**resize**(3);
	affiche_vector(v); // 0,1,2

capacity()	retourne le nombre courant d'emplacements mémoire réservés. C'est-à-dire le total de mémoire allouée en nombre d'éléments pour le conteneur. Attention, à ne pas confondre avec le nombre des éléments effectivement contenus retourné par size().
	ex : **vector<int>**v(10);
	cout<<"nombre elements : "<<v.**size**()<<endl;//10
	cout<<"capacite : "<<v.**capacity**()<<endl; // 10
	v.**resize**(12);
	cout<<"nombre elements : "<<v.**size**()<<endl; //12
	cout<<"capacite : "<<v.**capacity**()<<endl; // 15

Le parcours d'un « vector » peut aussi s'effectuer en utilisant des itérateurs (pointeurs) plutôt que le système d'indice. De ce fait, la classe « vector » est équipée avec toutes les méthodes les concernant :

Méthode	Description
begin()	retourne un itérateur « iterator » qui pointe sur le premier élément
end()	retourne un itérateur « iterator » qui pointe sur l'élément suivant le dernier
rbegin()	retourne un itérateur « reverse_iterator » qui pointe sur le premier élément de la séquence inverse (le dernier) ;
rend()	retourne un itérateur « reverse_iterator » qui pointe sur l'élément suivant le dernier dans l'ordre inverse (balise de fin, par exemple NULL)
cbegin()	retourne un itérateur « const_iterator » qui pointe sur le premier élément. L'élément pointé n'est alors accessible qu'en lecture et non en écriture. Il ne peut pas être modifié
cend()	retourne un itérateur « const_iterator » qui pointe sur ce qui suit le dernier élément (balise de fin, par exemple NULL). L'élément pointé n'est alors accessible qu'en lecture et non en écriture. Il ne peut pas être modifié
crbegin()	retourne un itérateur « const_reverse_iterator » qui pointe sur le premier élément de la séquence inverse (le dernier). L'élément pointé n'est accessible qu'en lecture et ne peut pas être modifié
crend()	retourne un itérateur « const_reverse_iterator » qui pointe sur la fin de la séquence inverse, avant le premier élément (balise de fin sens inverse, par exemple NULL). L'élément pointé n'est accessible qu'en lecture et ne peut pas être modifié

Exemple :

```
vector<int>vi(15, 7);  // 15 cases entières initialisées avec 7
vector<int>::iterator it;
   for (it=vi.begin(); it!=vi.end(); it++)
      cout<<*it<<"-";
   cout<<endl;
```

IV.1.2.3 Les vector et les fonctions

Passer un tableau dynamique en argument à une fonction est beaucoup plus simple que pour les tableaux statiques. Comme pour n'importe quel autre type, il suffit de mettre **vector<type>** en argument. Et c'est tout. Grâce à la fonction **size()**, il n'y a même pas besoin d'ajouter un deuxième argument pour la taille du tableau :

Exemple :

```
void afficheTableau(vector<int> tab)
{  for(int i = 0; i < tab.size(); i++)
     cout << tab[i] << endl;
}
```

✍ Si le tableau contient beaucoup d'éléments, le copier prendra du temps. Il vaut donc mieux utiliser un passage par référence constante const& pour optimiser la copie. Ce qui donne:

```
void afficheTableau(vector<int> const& tab)
{
      // ...
}
```

Dans ce cas, le tableau dynamique ne peut pas être modifié. Pour changer le contenu du tableau, il faut utiliser un passage par référence tout simple (sans le mot const donc). Ce qui donne :

```
void afficheTableau(vector<int>& tab)
{ … }
```

Pour appeler une fonction recevant un **vector** en argument, il suffit de mettre le nom du tableau dynamique comme paramètre entre les parenthèses lors de l'appel. Ce qui donne :

```
vector<int> tab(3,2);       //On crée un tableau de 3 entiers valant 2
afficheTableau(tab);       //On passe le tableau à la fonction afficherTableau()
```

Il est possible d'écrire une fonction renvoyant un vector.

```
vector<int> premierCarres(int n)
{   vector<int> tab(n);
   for(int i = 0; i < tab.size(); i++)
     tab[i] = i * i;
 return tab;
}
```

IV.2 Tableaux multidimensionnels dynamiques

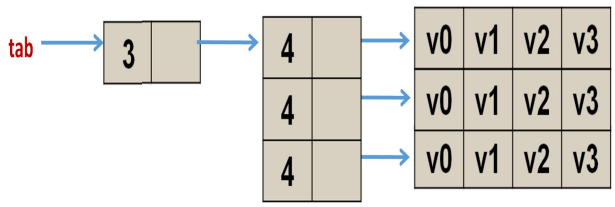
Notez qu'il est aussi possible de créer des tableaux multi-dimensionnels de taille variable en utilisant les **vector**s.

Pour un tableau 2D d'entiers, on devra écrire:

```
vector < vector<int> > tab;  // un vecteur de vecteurs :« tab » qui ne contient aucun élément
```

Pour dimensionner ce vecteur de vecteurs, il faudra le faire en 2 fois:

```
tab.resize(3); // 3 vecteurs de vecteurs vide pour l'instant
for (int ligne=0; ligne<tab.size(); ligne++ )
   tab [ligne].resize (4); // dimensionner chacun des vecteurs imbriqués
```



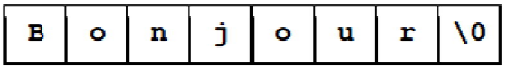
✍ Cela nous permet de créer un vecteur de 3 éléments désignant chacun 1 vecteur de 4 éléments. Finalement, on peut accéder aux valeurs dans les cases de du tableau en utilisant deux paires de crochets tab[i][j], comme pour les tableaux statiques. Il faut par contre s'assurer que cette ligne et cette colonne existent réellement.

Exemple :

```
vector<vector<int> > matrice;
matrice.push_back(vector<int>(5));   //On ajoute une ligne de 5 cases à notre matrice
matrice.push_back(vector<int>(3,4)); //On ajoute une ligne de 3 cases contenant chacune 4
matrice[0].push_back(8);   //Ajoute une case contenant 8 à la première ligne de la matrice
```

IV.3 Les strings sont des tableaux de caractères (lettres)

En C++, une chaîne de caractères n'est rien d'autre qu'un tableau de caractères, avec un caractère nul '\0' marquant la fin de la chaîne.
On peut par exemple représenter la chaîne « Bonjour » de la manière suivante :



Les chaînes de caractère peuvent être saisies au clavier et affichées à l'écran grâce aux objets habituels **cin** et **cout** .

IV.3.1 Déclaration

Pour définir une chaîne de caractères en langage C, il suffit de définir un tableau de caractères. Le nombre maximum de caractères que comportera la chaîne sera égal au nombre d'éléments du tableau moins un (réservé au caractère de fin de chaîne).

Exemple :

```
char nom[20], prenom[20];    // 19 caractères utiles
char adresse[3][40];    // trois lignes de 39 caractères utiles
char texte[10] = {'B','o', 'n', 'j','o', 'u','r', '\0'};    // ou : char texte[10] = "Bonjour";
```

✍ On peut utiliser un **vector** si l'on souhaite changer la longueur du texte :

```
vector<char> texte;
```

✍ En théorie, on pourrait donc se débrouiller en utilisant des tableaux statiques ou dynamiques de **char** à chaque fois que l'on veut manipuler du texte. Mais ce serait fastidieux. C'est pour ça que les concepteurs du langage ont décidé de cacher tout ces mécanismes dans une boîte fermée (Objet) en utilisant la classe **string**. Cette dernière, propose en fait une encapsulation de la chaîne de caractères C. La bibliothèque standard du C++ propose d'autres types de chaînes de caractères, notamment celles qui sont capables de gérer un encodage comme l'UTF-8 où un caractère est stocké sur plusieurs octets.

IV.3.2 Création d'objets « string »

La création d'un objet ressemble beaucoup à la création d'une variable classique comme **int** ou **double**:

```
#include <iostream>
#include <string>    // Obligatoire pour pouvoir utiliser les objets string
using namespace std;
int main()
{
   string maChaine;  //Création d'un objet 'maChaine' de type string
   return 0;
}
```

IV.3.3 Instanciation et initialisation de chaînes

Pour initialiser notre objet au moment de la déclaration (et donc lui donner une valeur !), il y a plusieurs possibilités:

```
int main()
{   string maChaine("Bonjour !");     //Création d'un objet 'maChaine' de type string et initialisation
    String  s3 = "chaine 3";
    return 0;
}
```

IV.3.4 Accès à un caractère

On manipule une chaîne C++ comme une chaîne C : on peut utiliser les crochets pour accéder à un élément même si la chaîne C++ n'est pas un tableau. Nous verrons plus tard comme on peut faire cela.

```
// lecture d'un caractère
cout << s2[3];        // 4eme caractère
cout << s2.at(3);   // 4eme caractère
// modification de caractère
s2[2]    = 'A';
s2.at(3) = 'B';
```

IV.3.5 Concaténation de chaînes

Cette opération permet d'assembler deux chaines de caractères:

```
#include <iostream>
#include<string>
 using namespace std;
 int main (void)
{   string s1, s2, s3;
     cout << "Tapez une chaine : "; getline (cin, s1);
     cout << "Tapez une chaine : "; getline (cin, s2);
     s3 = s1 + s2;
     cout << "Voici la concatenation des 2 chaines :" << endl;
     cout << s3 << endl;
  return 0;
}
```

- Par défaut, lorsqu'on saisit une chaîne de caractères en utilisant cin, le séparateur est l'espace : cela empêche de saisir une chaîne de caractères comportant un espace.

- La fonction **getline(iostream &,string)** permet de saisir une chaîne de caractères en utilisant le passage à la ligne comme séparateur : notre chaîne de caractères peut alors comporter des espaces.

- On peut utiliser une autre syntaxe de concaténation: s1.**append** (s2) qui est équivalente à s1 = s1+s2

IV.3.6 Quelques méthodes utiles du type « string » [16]

IV.3.6.1 Méthode size()

La méthode **size()** permet de connaître la longueur de la chaîne actuellement stockée dans l'objet de type **string**.

Exemple :

```
int main()
{   string maChaine("Bonjour !");
   cout << "Longueur de la chaine : " << maChaine.size();
    return 0;
}
```

```
Longueur de la chaine : 9
```

Affichage de l'exécution

IV.3.6.2 Méthode « erase() »

Cette méthode très simple supprime tout le contenu de la chaîne :

Exemple :

```
int main()
{    string chaine("Bonjour !");
    chaine.erase(0, 4);   // efface les 4 premiers caractères
    cout << "La chaine contient : " << chaine << endl;
    chaine.erase();
    cout << "La chaine contient : " << chaine << endl;
    return 0;
}
```

Affichage de l'exécution:

```
La chaine contient :our !
La chaine contient :
```

IV.3.6.3 Méthode « substr() »

Une autre méthode peut se révéler utile: **substr()**. Elle permet d'extraire une partie de la chaîne stockée dans un string.

Syntaxe : string **substr**(size_type index, size_type num = npos);

- **Index**: permet d'indiquer à partir de quel caractère on doit couper (ce doit être un numéro de caractère).
- **Num** : permet d'indiquer le nombre de caractères que l'on prend. Par défaut, la valeur est ***npos***, ce qui revient à prendre tous les caractères qui restent. Si vous indiquez 2, la méthode ne renverra que 2 caractères.

Exemple:

```
int main()
{    string chaine("Bonjour !");
    cout << chaine.substr(3) << endl;
    return 0;
}
```

```
Jour !
```

Affichage de l'exécution

```
int main()
{   string chaine("Bonjour !");
    cout << chaine.substr(3, 4) << endl;
    return 0;
}
```

```
Jour
```

Affichage de l'exécution

✍ On a demandé à prendre 4 caractères en partant du caractère n°3, ce qui fait qu'on a récupéré « jour ».

IV.3.6.4 Méthode «c_str() »

Cette méthode permet de renvoyer un pointeur vers le tableau de char que contient l'objet de type string.

- **Transformation de chaîne de type C en string:** on peut utiliser le constructeur string(char *) ou l'affectation grâce au symbole = d'un char * vers une string.
- **Transformation d'un string en chaîne de type C :** il suffit d'utiliser la méthode : **c_str()** qui renvoie un char * qui est une chaîne de type C.

Exemple:

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

int main (void)
{  string s1, s2;
    char c1 []= "BONJOUR";
    const char * c2;
    s1 = c1;
    cout << s1 << endl;
    s2 = "AU REVOIR";
    c2 = s2.c_str();
    cout << c2 << endl;
  return 0;
 }
```

```
BONJOUR
AU REVOIR
```

Affichage de l'exécution

✍ Dans cet exemple, c1 est un tableau de 8 char contenant la chaîne "BONJOUR " sans oublier le caractère de fin de chaîne '\0'.

✍ Le pointeur c2 est un pointeur vers un tableau non modifiable de char.

✍ Les variables s1 et s2 sont des string. On peut affecter directement s1=c1 : le tableau de char sera transformé en string.

✍ Dans c2, on peut récupérer une chaîne « de type C » identique à notre string en écrivant c2=s2.**c_str()**. On peut transformer aisément un string en tableau de char et inversement.

IV.3.6.5 Méthode «istr() »

- Pour transformer une chaîne en double ou en int, il faut transformer la chaîne en flot de sortie caractères : il s'agit d'un **istringstream**. Ensuite, nous pourrons lire ce flot de caractères en utilisant les opérateurs usuels >>.
- La méthode **eof()** sur un **istringstream** permet de savoir si la fin de la chaîne a été atteinte. Chaque lecture sur ce flot grâce à l'opérateur >> renvoie un booléen qui nous indique d'éventuelles erreurs.

Exemple :

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main (void)
 {
  string s;
  cout << "Tapez une chaine : "; getline (cin, s);
  istringstream istr(s);
  int i;

  if (istr >> i) cout << "VOUS AVEZ TAPE L'ENTIER " << i << endl;
  else cout << "VALEUR INCORRECTE" << endl;
  return 0;
 }
```

```
Tapez une chaîne : 12345
VOUS AVEZ TAPE L'ENTIER 12345
```

Affichage de l'exécution

✍ Dans cet exemple, ***s*** est une chaîne : on saisit une chaîne au clavier en utilisant getline(cin,s). On crée ensuite un **istringstream** appelé ***istr*** et construit à partir de ***s***.

✍ On peut lire un entier ***i*** à partir de ***istr*** en utilisant : istr>>i . (istr>>i) renvoie true si un entier valide a pu être lu et renvoie false sinon. De la même manière, on pourrait lire des données d'autres types, double par exemple.

IV.3.6.6 Autres opérations : insert, replace, find

```
s1.insert(3, s2);   // insère s2 à la position 3
s3.replace(2,3,s1);   // remplace la portion de s3 définie de la position 2 à 5 par la chaîne s1
unsigned int i = s.find("trouve", 4);  // recherche "trouve" à partir de la 4ème position, renvoie
                                       //   std::string::npos le cas échéant
```