CHARLES CARNUS
LYCÉE TECHNIQUE
ENSEIGNEMENT SUPÉRIEUR

Programmation

C++ : Pointeurs et fonctions

Prof : Kamal Boudjelaba

15 mars 2024

B T S S N

LTP Charles Carnus
Avenue de Saint Pierre - Bourran - 12000 RODEZ, France

IR–EC

Table des matières

Liste des figures

Liste des tableaux

1. Les pointeurs

L'une des principales caractéristiques du langage C++ est le concept de pointeur. Les pointeurs sont utiles pour allouer de la mémoire à des tableaux dont la taille n'est pas connue au moment de la compilation du code. Ils ont aussi leur utilité lors de l'écriture de fonctions permettant de répéter la même opération sur des variables différentes.

Définition 1.1 : Les pointeurs

Un pointeur est une variable qui contient l'adresse d'une autre variable (objet).

Lorsque une variable est déclarée, de la mémoire est allouée à cette variable, et l'emplacement de cette mémoire ne variera pas tout au long de l'exécution du code.
En plus des types de données tels que les entiers et les nombres à virgule flottante, on peut également déclarer des variables de pointeur qui sont des variables qui stockent les adresses d'autres variables, c'est-à-dire l'emplacement dans la mémoire de l'ordinateur.

1.1 Case mémoire et adresse

— Tout objet (variable, fonction ...) manipulé par l'ordinateur est stocké dans sa mémoire, constituée d'une série de cases.

— Pour accéder à un objet (au contenu de la case mémoire dans laquelle cet objet est enregistré), il faut connaître le numéro de cette case. Ce numéro est appelé l'adresse de la case mémoire.

— Lorsqu'on utilise une variable ou une fonction, le compilateur utilise l'adresse de cette dernière pour y accéder.

Adresse mémoire					

Figure 1. Pointeur

1.2 Syntaxe : Déclarer un pointeur

Pour déclarer un pointeur, on procède de la même manière que pour les variables, c.à.d. déclarer :

- le type
- le nom, précédé par $*$

```
int *pointeur;
```

On peut aussi utiliser cette syntaxe :
`int* pointeur;`
Inconvenient : ne permet pas de déclarer plusieurs pointeurs sur la même ligne.

```
int* p1, p2, p3, p4;
```

Seul $p1$ sera un pointeur, les autres variables seront des entiers standards.

Exemples

```
string *pointeur1;
//Un pointeur qui peut contenir l'adresse d'une chaîne de caractères

vector<int> *pointeur2;
//Un pointeur qui peut contenir l'adresse d'un tableau de nombres entiers

double* p_x;
// p_x est un pointeur vers une variable à virgule flottante double précision

int* p_i;
// p_i est un pointeur vers une variable entière
```

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    cout << &x << endl;
    return 0;
}
```

En C++, le symbole pour obtenir l'adresse d'une variable est &. Pour afficher l'adresse de la variable x, on doit écrire $\&x$.
Après exécution du code, la console affiche `0x7ffeefbff538`, qui correspond à l'adresse (en hexadécimal) de la case mémoire contenant la variable x.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    cout << &x << endl;
    int *p_x = nullptr; // ou int *p_x(0);
    p_x = &x;
    cout << p_x << endl;
    return 0;
}
```

Important : Les pointeurs doivent être déclarés en les initialisant à 0.
La console affiche :

```
0x7ffeefbff538
0x7ffeefbff538
```

L'adresse de x est sauvegardée dans le pointeur p_x.
On dit alors que le pointeur p_x pointe sur x.

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

int main()
{
    int a, b, c;
    int *x(nullptr), *y(nullptr);

    a = 98;
    x = &a;
    c = *x + 5;
    y = &b;
    *y = a + 10;

    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    return 0;
}
```

La console affiche :

```
b = 108
c = 103
```

- a est initialisé à 98.
- x = $\&a$; affecté à x l'adresse de a.
  x est un pointeur vers a.
- $*x$ est la variable pointée pas x, c.à.d a (=98).
- c = $*x$ + 5 ; permet de transférer $98+5=103$ dans la variable c.
- y = $\&b$; permet de mettre dans la variable y l'adresse de la variable b. y un pointeur vers b.
  $a+10=98+10=108$.
- $*y$ = a + 10 ; permet de transférer dans la variable pointée par y la valeur de a + 10 = 108.
  On stocke 108 dans b de manière indirecte via le pointeur y.
- Affichage des valeurs de b et c.

Si une variable `p_x` a été déclarée comme pointeur vers un nombre double, alors il est important de distinguer :

(i) l'emplacement de la mémoire vers lequel pointe ce pointeur (noté `p_x`) ; et

(ii) le contenu de cette mémoire (noté `*p_x`).

L'opérateur astérisque (`*`) dans `*p_x` est appelé déréférencement de pointeur et peut être considéré comme l'opposé de l'opérateur `&`.
Le code ci-dessous montre comment des pointeurs vers des variables réelles peuvent être combinés avec des variables.

```
1 double y, z; // y, z stockent les nombres en double précision
2 double* p_x; // p_x stocke l'adresse d'un nombre double
3
4 z=3.0;
5 p_x = &z; // p_x stocke l'adresse de z
6 y = *p_x + 1.0; // *p_x est le contenu de la mémoire p_x, c'est-à-dire la valeur de z
```

Avertissements sur l'utilisation des pointeurs

Un pointeur de variable ne doit pas être utilisé avant d'avoir d'abord reçu une adresse valide. Par exemple, le fragment de code suivant peut entraîner des problèmes difficiles à localiser.

```
1 double* p_x ; // p_x peut stocker l'adresse d'un nombre double - on ne connait pas l'adresse
                                                    encore
2
3 *p_x = 1,0 ; // essaie de stocker la valeur 1.0 dans un emplacement mémoire non spécifié
```

Dans le code ci-dessus, on n'a pas spécifié l'emplacement de la variable double vers laquelle pointe `p_x`. Il peut donc pointer n'importe où dans la mémoire de l'ordinateur. Changer le contenu d'un emplacement non spécifié dans la mémoire d'un ordinateur, comme cela est fait à la ligne 5 du code ci-dessus, a clairement le potentiel de causer des problèmes qui peuvent être difficiles à localiser. Ce problème peut être évité en utilisant le nouveau mot-clé comme indiqué ci-dessous pour allouer une adresse mémoire valide à `p_x`, et le mot-clé `delete` qui libère cette mémoire pour qu'elle soit utilisée par d'autres parties du programme lorsque cette mémoire n'est plus requise.

```
1 double* p_x ; // p_x stocke l'adresse d'un nombre double
2
3 p_x = new double ; // attribue une adresse à p_x
4 *p_x = 1,0 ; // stocke 1.0 en mémoire avec adresse p_x
5
6 delete p_x ; // libère de la mémoire pour une réutilisation
```

Une autre raison d'utiliser les pointeurs avec précaution est indiquée dans le code ci-dessous. La première fois que y est imprimé (à la ligne 5) il prend la valeur 3 : la deuxième fois que y est imprimé (à la ligne 7) il prend la valeur 1 même si y n'est pas explicitement modifié dans le code entre ces deux lignes. En effet, la ligne entre les instructions `std::cout`, ligne 6, a modifié la valeur de y, peut-être involontairement, en utilisant la variable de pointeur `p_x` (qui contient l'adresse de y) pour modifier la valeur de y.

```
1 double y;
2 double* p_x;
3 y = 3.0;
4 p_x = &y;
5 std::cout << "y = " << y << "\n";
6 *p_x = 1.0; // Cela change la valeur de y
7 std::cout << "y = " << y << "\n";
```

Une situation où le contenu de la même variable peut être accédé en utilisant des noms différents, comme dans le code ci-dessus, est connue sous le nom d'aliasing. En C++, cela est plus susceptible de se produire lorsque des pointeurs sont impliqués, soit lorsque deux pointeurs pointent la même adresse en mémoire, soit lorsqu'un pointeur fait référence au contenu d'une autre variable. Lorsqu'un ou plusieurs pointeurs permettent d'accéder à la même variable en utilisant des noms différents, l'aliasing est appelé aliasing de pointeur.

1.3 Allocation dynamique de mémoire pour les tableaux

L'une des principales utilisations des pointeurs est l'allocation dynamique de mémoire pour stocker des tableaux. Dans le cours sur les tableaux, on a expliqué comment les tableaux pouvaient être déclarés lorsque la taille du tableau était connue à l'avance. Cependant, on ne pas toujours la taille des tableaux dans un programme lorsqu'on compile le code. L'utilisation de pointeurs pour allouer dynamiquement de la mémoire aux tableaux évite des problèmes, car on n'a pas besoin de connaître la taille du tableau au moment de la compilation.

Tableau unidimensionnel

Pour utiliser des pointeurs pour créer un tableau unidimensionnel de nombres réels de longueur 10 appelé `x`, on utilise le code suivant :

```
double* x;
x = new double[10];
```

Les éléments du tableau sont alors accessibles exactement de la même manière que si le tableau avait été créé en utilisant le type de déclaration introduit dans le cours sur les tableaux. Dans l'allocation dynamique de mémoire pour le tableau à l'aide du pointeur `x` ci-dessus, `x` stocke l'adresse du premier élément du tableau. Cela peut être vu en affichant à la fois le pointeur `x` et l'adresse du premier élément du tableau, comme indiqué ci-dessous :

```
std::cout << x << "\n";
std::cout << &x[0] << "\n"; // affiche la même valeur
```

La mémoire allouée à `x` peut être, et doit être, désallouée en utilisant l'instruction `delete[] x;` lorsque ce tableau n'est plus nécessaire.

Un exemple de code qui utilise la mémoire allouée dynamiquement pour les tableaux est présenté ci-dessous. Ce code crée deux tableaux, `x` et `y`, tous deux de taille 10. Les éléments de `x` sont ensuite affectés manuellement. Les éléments de `y` sont alors définis pour être le double de la valeur de l'élément correspondant de `x`. Enfin, toute la mémoire allouée est supprimée.

```
#include <iostream>

int main(int argc, char* argv[])
{
    double* x;
    double* y;
    x = new double [10];
    y = new double [10];

    for (int i=0; i<10; i++)
    {
        x[i] = ((double)(i));
        y[i] = 2.0*x[i];
    }

    delete[] x;
    delete[] y;

    return 0;
}
```

Matrice

La mémoire pour les matrices peut également être allouée dynamiquement. Par exemple, pour créer un tableau à deux dimensions de nombres réels avec 5 lignes et 3 colonnes appelé `A`, on utilise le code suivant :

```
1 int rows = 5, cols = 3;
2 double** A;
3 A = new double* [rows];
4 for (int i=0; i<rows; i++)
5 {
6     A[i] = new double [cols];
7 }
```

Le tableau peut alors être utilisé exactement de la même manière que s'il avait été créé en utilisant la déclaration `double A[5][3];`

Lors de l'allocation dynamique de mémoire pour la matrice dans le code ci-dessus, la variable `A`, qui a été déclarée à l'aide de la ligne 2, a les propriétés suivantes après l'exécution du fragment de code :
- each `A[i]` est un pointeur et contient l'entête `A[i][0];` et
- A contient l'adresse du pointeur `A[0]`.

Ainsi, la variable `A` est un tableau de pointeurs, ce qui explique les deux astérisques de la ligne 2 du code. La ligne 3 spécifie que `A` est un pointeur vers un tableau de pointeurs vers des nombres réels, et que ce tableau est de taille lignes. La boucle `for` spécifie ensuite que chaque pointeur du tableau lui-même pointe vers un tableau de nombres réels de longueur cols. Cela a pour effet que `A[i]` - qui est un pointeur - stocke l'adresse de l'entrée `A[i][0]`, c'est-à-dire la première entrée de la `ligne i`.
Comme c'était le cas pour les vecteurs, il est important de désallouer dynamiquement la mémoire allouée à une matrice lorsqu'elle n'est plus nécessaire. La mémoire allouée pour la matrice `A` peut être libérée en utilisant le code suivant :

```
1 for (int i=0; i<rows; i++)
2 {
3     delet[] A[i];
4 }
5 delete[] A;
```

2. Les références

Dans la section 1, on a montré l'utilisation de pointeurs pour permettre aux modifications apportées à une variable dans une fonction d'avoir un effet en dehors de la fonction, et montré comment cela pourrait être utilisé pour permettre à une fonction de renvoyer plus d'une variable. Une alternative à l'utilisation de pointeurs est d'utiliser des variables de référence : ce sont des variables qui sont utilisées à l'intérieur d'une fonction et qui sont un nom différent pour la même variable que celle envoyée à une fonction. Lors de l'utilisation de variables de référence, toute modification à l'intérieur de la fonction aura un effet en dehors de la fonction. Ceux-ci sont beaucoup plus faciles à utiliser que les pointeurs : il suffit d'inclure le symbole `&` devant le nom de la variable dans la déclaration de la fonction et du prototype — cela indique que la variable est une variable de référence. C'est en fait le cas où les références se comportent comme des pointeurs, mais sans que le programmeur ait à convertir en une adresse avec `&` sur l'appel de la fonction et sans avoir à déréférencer à l'intérieur de la fonction — elles fournissent une syntaxe pour alléger le fardeau du programmeur.

- Une référence peut être vue comme un alias d'une variable (utiliser la variable ou une référence à cette variable est équivalent).
- On peut modifier le contenu de la variable en utilisant une référence.
- Une référence ne peut être initialisée qu'une seule fois : à la déclaration. Toute autre affectation modifie en fait la variable référencée.
- Une référence ne peut donc référencer qu'une seule variable.
- Les références sont principalement utilisées pour passer des paramètres aux fonctions.

2.1 Syntaxe : déclarer une référence

```
type &reference = identificateur;
```

Exemples

```
int x = 0;
int &r_x = x;        // Référence sur la variable x.
r_x = r_x + x;       // Double la valeur de x (et de r_x).
```

```
#include <iostream>
using namespace std;
int main()
{
    int a = 98, b = 78, c;
    int &x = a;
    c = x + 5;  // équivalent à :  c = a + 5;
    int &y = b;
    y = a + 10; // équivalent à :  b = a + 10;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    return 0;
}
```

- `int &x = a;` permet de déclarer une référence x vers la variable a.
- `c = x + 5;` permet donc de transférer 103 dans la variable c.
- `int &y = b;` permet de déclarer une référence y vers la variable b.
- `y = a + 10;` permet de transférer 108 dans la variable b.

La console affiche :

```
b = 108
c = 103
```

Exemple 2.1: Calcul de la partie réelle et imaginaire d'un nombre complexe

On donne un nombre complexe sous forme polaire, $z = re^{i\theta}$, on veut écrire une fonction qui renvoie la partie réelle, notée par la variable `x`, et la partie imaginaire, notée par la variable `y`, de ce nombre.
Donc on utilise une fonction pour calculer les parties réelles et imaginaires d'un nombre complexe donné sous forme polaire en utilisant des références au lieu des pointeurs (voir section 3.4 pour la version avec des pointeurs).

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

void CalculReelEtImaginaire(double r, double theta, double& reel, double& imaginaire);

int main(int argc, char* argv[])
{
    double r = 3.4;
    double theta = 1.23;
    double x, y;

    CalculReelEtImaginaire(r, theta, x, y);

    std::cout << "Partie réelle = " << x << "\n";
    std::cout << "Partie imaginaire = " << y << "\n";

    return 0;
}

void CalculReelEtImaginaire(double r, double theta, double& reel, double& imaginaire)
{
    reel = r*cos(theta);
    imaginaire = r*sin(theta);
}
```

3. Les fonctions

3.1 Introduction

Un bloc est n'importe quel morceau de code entre accolades. Une variable, lorsqu'elle est déclarée à l'intérieur d'un bloc, peut être utilisée dans tout ce bloc, mais uniquement à l'intérieur de ce bloc. Ceci est démontré dans le code ci-dessous. A la ligne 9, on tente d'utiliser la variable `j` alors qu'elle n'est déclarée - et donc disponible - que dans le bloc entouré d'accolades aux lignes 4 et 8. Dans le langage des programmeurs, "la portée de `j` est le bloc entre lignes 4 et 8". Si on essaye d'utiliser le fragment de code ci-dessous, le compilateur signalerait cette tentative d'utilisation de `j` comme une erreur : `j` est dit hors de portée à la ligne 9.

```
1  {
2      int i;
3      i=5; //OK
4      {
5          int j;
6          i = 10; // OK
7          j = 10; // OK
8      }
9      j = 5; // incorrect - j nest pas déclaré ici
10 }
```

Le même nom de variable peut être utilisé pour une variable déclarée à la fois à l'intérieur d'un bloc - appelée variable locale - et en dehors de la portée de toute fonction (y compris la fonction principale) - appelée variable globale. Ces deux variables sont accessibles à l'intérieur du bloc comme indiqué dans le code ci-dessous, en utilisant l'exemple d'une variable globale et d'une variable locale appelée `i`. De plus, on peut définir une variable `j` à la fois dans le bloc externe et dans le bloc interne : à l'intérieur du bloc interne la valeur de `j` stockée par la variable déclarée dans le bloc externe n'est pas accessible. La déclaration multiple de `i` et `j` dans le code ci-dessous est une mauvaise pratique de programmation, car elle peut clairement prêter à confusion. En fait, puisque la portée des variables est si importante, on suggère que les variables soient déclarées uniquement dans le bloc où elles sont nécessaires, à proximité de leur première utilisation. Cette déclaration multiple de variables est connue sous le nom d'occultation de variable et on peut l'éviter en activant les "shadow warnings" dans le compilateur.

```
1  #include <iostream>
2  int i = 5; // global i
3
4  int main(int argc, char* argv[])
5  {
6      int j = 7;
7      std::cout << i << "\n";
8      {
9          int i = 10, j = 11;
10         std::cout << i << "\n";      // la valeur locale de i est 10
11         std::cout << ::i << "\n";    // la valeur globale de i est 5
12         std::cout << j << "\n";      // la valeur de j ici est 11
13         // L'autre j (valeur 7) est inaccessible
14     }
15     std::cout << j << "\n";          // la valeur de j ici est 7
16     return 0;
17 }
```

3.2 Les fonctions

Les fonctions ont la forme suivante :

```
1  type nom_de_la_fonction(arguments)
2  {
3      //Instructions effectuées par la fonction
4  }
```

— **type** : indique le type de variable renvoyée par la fonction (string, int, double ...)

— `nom_de_la_fonction` : permet de donner un nom à la fonction

— `arguments` : les différentes variables d'entrée de la fonction

```
1 int multiplicationDix(int nbre)
2 {
3     int res(nbre * 10);
4
5     return res;
6 }
```

Exemple 3.1: Calcul du minimum

Un programme simple qui écrit et utilise une fonction pour déterminer la valeur minimale de deux variables réelles `x` et `y`, et la stocke dans la variable réelle `valeur_min` est illustré ci-dessous. On note le prototype de fonction qui est la ligne 3 du code ci-dessous. Le prototype de fonction indique au compilateur quelles variables d'entrée sont requises et quelle variable, le cas échéant, est renvoyée. Dans l'exemple ci-dessous, le prototype de fonction explique que plus tard dans le code, il y aura une fonction appelée `CalculMinimum` qui nécessite deux variables réelles en entrée et renvoie une variable réelle. Le prototype de fonction peut être considéré comme étant similaire à la déclaration d'une variable. Les noms de variables `a` et `b` dans le prototype sont ignorés par le compilateur et n'ont pas à être inclus, mais leur inclusion peut clarifier le programme. On note aussi que le prototype de la fonction se termine par un point-virgule.
Les lignes 15 à 29 du code contiennent les instructions qui exécutent les tâches requises par la fonction. Ce code commence par une ligne de code identique au prototype de fonction (y compris les noms de variables) sans le point-virgule. Après cela, il y a un bloc de code qui se termine par une instruction `return` qui renvoie la valeur requise au point du code d'où cette fonction a été appelée. On note qu'il n'est pas nécessaire de déclarer les variables `a` et `b` dans la fonction - la déclaration de la ligne 15 l'a déjà fait. Les variables telles que `minimum` qui sont utilisées à l'intérieur de la fonction mais ne font pas partie du prototype de la fonction doivent être déclarées dans le bloc fonction. La ligne 8 montre comment appeler une fonction : les variables entre parenthèses (`x` et `y` dans ce cas) sont envoyées à la fonction et sont connues comme les arguments de la fonction. La variable renvoyée par la fonction est stockée en tant que `valeur_min`.

```
1  #include <iostream>
2
3  double CalculMinimum(double a, double b);
4
5  int main(int argc, char* argv[])
6  {
7      double x = 4.0, y = -8.0;
8      double valeur_min = CalculMinimum(x, y);
9      std::cout<<"Le minimum de "<< x << " et " << y << " est " << valeur_min << "\n";
10
11
12     return 0;
13 }
14
15 double CalculMinimum(double a, double b)
16 {
17     double minimum;
18     if (a < b)
19     {
20         minimum = a;
21     }
22     else
23     {
24         //a>=b
25         minimum = b;
26     }
27
28     return minimum;
29 }
```

Exemple 3.2: Multiplication par 10

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

int multiplicationDix(int nbre)
{
    int res(nbre * 10);

    return res;
}

int main()
{
    int a(5),b(9);
    cout << "Valeur de a : " << a << endl;
    cout << "Valeur de b : " << b << endl;
    b = multiplicationDix(b); //Appel de la fonction
    cout << "Valeur de a : " << a << endl;
    cout << "Valeur de b : " << b << endl;

    return 0;
}
```

Exemple 3.3: Fonction `void`

Dans certains cas, on ne veut pas qu'une fonction renvoie une variable : de telles fonctions peuvent être prototypées comme une fonction `void`. Le code ci-dessous contient un exemple de fonction qui affiche un message informant un candidat s'il a réussi ou non un examen. Cette fonction nécessite deux variables entières en entrée : la première d'entre elles contient la note qu'un candidat a obtenue ; la seconde contient la note de passage à l'examen.

```cpp
#include <iostream>

void AffichePassOuNon(int note, int notePassage);

int main(int argc, char* argv[])
{
    int note = 39, note_passage = 30;
    AffichePassOuNon(note, note_passage);

    return 0;
}

void AffichePassOuNon(int note, int notePassage)
{
    if (note >= notePassage)
    {
        std::cout << "Pass - Félicitations!\n";
    }
    else
    {
        // note < note_Passage
        std::cout << "Non - Bonne chance la prochaine fois\n";
    }
}
```

Une fonction ne peut modifier que la valeur d'une variable envoyée à une fonction à l'intérieur de cette fonction : les modifications apportées à l'intérieur de la fonction n'auront aucun effet sur cette variable après l'exécution de la fonction et le code continue à exécuter des instructions dans le bloc où la fonction a été appelé. En effet, une copie est faite de toute variable envoyée à une fonction, et c'est cette copie de la variable, et non la variable d'origine, qui est modifiée à l'intérieur de la fonction. Par exemple, la fonction suivante n'a aucun effet sur la variable `x` en dehors

de la fonction, même si la valeur de `x` est modifiée à l'intérieur de la fonction.

```cpp
#include <iostream>

void SansEffet(double x);

int main(int argc, char* argv[])
{
    double x = 2.0;
    SansEffet(x);
    std::cout << x << "\n"; // affichera 2.0

    return 0;
}

void SansEffet(double x)
{
    // x prend ici la valeur 2.0
    x += 1.0;
    // x prend ici la valeur 3.0
}
```

Exemple 3.4: A compléter (Fonction x^n avec $x \in \mathbb{R}$ et $n \in \mathbb{N}$)

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

double puissance(double x, int n) {
  //algorithme de calcul de x^n (à coder)
}

int main() {
  double x;
  int n;
  cout << "Donner x et n : ";
  cin >> x >> n;
  cout << x << "^" << n << " = " << puissance(x, n) << "\n";
}
```

3.3 Renvoi de variables de pointeur à partir d'une fonction

Les fonctions peuvent également être utilisées pour renvoyer des variables de pointeur, comme indiqué dans le code ci-dessous. Dans ce cas, on a écrit une fonction qui alloue dynamiquement de la mémoire pour une matrice et renvoie le pointeur vers la mémoire allouée. Le tableau peut alors être utilisé comme si la mémoire était allouée dans la fonction main, comme démontré aux lignes 8 et 9. Parce que chaque nouveau nécessite une suppression correspondante, on a évité les fuites de mémoire en fournissant également une fonction appelée `FreeMatrixMemory` pour libérer la mémoire créée dans `AllocateMatrixMemory`.

```
1  #include <iostream>
2  using namespace std;
3
4  double** AllocateMatrixMemory(int numRows, int numCols);
5  void FreeMatrixMemory(int numRows, double** matrix);
6
7  int main(int argc, char* argv[])
8  {
9      double** A;
10     A = AllocateMatrixMemory(5, 3);
11     A[0][1] = 2.0;
12     A[4][2] = 4.0;
13     FreeMatrixMemory(5, A);
14     return 0;
15 }
16
17 // Function to allocate memory for a matrix dynamically
18 double** AllocateMatrixMemory(int numRows, int numCols)
19 {
20     double** matrix;
21     matrix = new double* [numRows];
22     for (int i=0; i<numRows; i++)
23     {
24         matrix[i] = new double [numCols];
25     }
26     return matrix;
27 }
28
29 // Function to free memory of a matrix
30 void FreeMatrixMemory(int numRows, double** matrix)
31 {
32     for (int i=0; i<numRows; i++)
33     {
34         delete[] matrix[i];
35     }
36     delete[] matrix;
37 }
```

3.4 Utilisation de pointeurs comme arguments de fonction

Dans certains cas, on souhaite que les modifications apportées à une variable à l'intérieur d'une fonction aient un effet en dehors d'une fonction. On reprend l'exemple 1, qui à partir d'un nombre complexe donné sous forme polaire, $z = re^{i\theta}$, on veut écrire une fonction qui renvoie la partie réelle, notée par la variable `x`, et la partie imaginaire, notée par la variable `y`, de ce nombre.
On a noté précédemment qu'une fonction ne peut renvoyer qu'une seule variable, et on ne peut donc pas renvoyer à la fois la variable `x` et la variable `y`. Il serait donc utile d'inclure les variables `x` et `y` dans l'appel de la fonction. Cependant, cela ne fonctionnerait pas non plus, car les valeurs attribuées à ces variables n'auraient aucun effet en dehors de la fonction. Heureusement, les pointeurs fournissent un moyen de contourner ce problème. Au lieu d'envoyer les variables `x` et `y` à la fonction, on envoie les adresses de ces variables à la fonction. Lorsque la fonction est appelée, des copies sont faites des adresses de ces variables, et ce sont ces copies qui sont envoyées à la fonction. Les modifications apportées à ces adresses n'auront aucun effet en dehors de la fonction car on travaille avec une copie de ces adresses. Cependant, on peut changer le contenu de la variable sans changer l'adresse en déréférencant le pointeur, et cela aura un effet en dehors de la fonction. Ceci est montré dans le code ci-dessous :

```
1  #include <iostream>
2  #include <cmath>
3
4  void CalculReelEtImaginaire(double r, double theta,
5                              double* pReel,
6                              double* pImaginaire);
7
8  int main(int argc, char* argv[])
9  {
10     double r = 3.4;
11     double theta = 1.23;
12     double x, y;
13     CalculReelEtImaginaire(r, theta, &x, &y);
14     std::cout << "Partie réelle = " << x << "\n";
15     std::cout << "Partie imaginaire = " << y << "\n";
16
17     return 0;
18 }
19
20 void CalculReelEtImaginaire(double r, double theta,
21                             double* pReel,
22                             double* pImaginaire)
23 {
24    *pReel = r*cos(theta);
25    *pImaginaire = r*sin(theta);
26 }
```

Les lignes 4 à 6 du code sont vraiment censées être une longue ligne, donnant le prototype de fonction de `CalculReelEtImaginaire`. Étant donné que la ligne est longue, on l'a divisée en plusieurs lignes et indenté les lignes de continuation pour plus de clarté. Le prototype répertorie les arguments de la fonction. Les deux premiers arguments sont des variables réelles représentant le module (notée `r`) et l'argument (noté `thêta`) du nombre complexe spécifié. Les troisième et quatrième arguments sont des pointeurs vers - c'est-à-dire les adresses de - la partie réelle et la partie imaginaire du nombre complexe. À la ligne 12, on déclare des réels (`double`) `x` et `y` qui représentent les parties réelle et imaginaire du nombre complexe. Pour utiliser la fonction `CalculReelEtImaginaire`, on envoie les adresses de ces variables à la fonction. Une copie de ces adresses est faite, et ce sont ces copies qui sont utilisées dans la fonction aux lignes 20–26. Cependant, ces copies se réfèrent à la même mémoire que les variables d'origine `x` et `y`, et c'est donc dans cette mémoire que les résultats des calculs des lignes 24 et 25 sont stockés.

3.5 Envoi de tableaux aux fonctions

Lors de l'envoi de tableaux à des fonctions, que la mémoire ait été allouée dynamiquement ou non, il convient de noter que c'est l'adresse du premier élément du tableau qui est envoyée à la fonction. Tout comme l'envoi du pointeur vers une variable à une fonction, les modifications apportées à cette adresse n'auront pas d'effet sur le code à partir duquel cette fonction est appelée : cependant, le contenu de cette adresse, c'est-à-dire le contenu du tableau, peut être changé. Ainsi, toute modification apportée à un tableau à l'intérieur d'une fonction aura un effet lorsque cette variable sera utilisée par la suite en dehors de la fonction.
On commence par montrer comment envoyer à une fonction des tableaux dont la taille est connue au moment de la compilation. Ceci est indiqué dans le code ci-dessous. On note qu'on n'a pas à spécifier la taille du premier index d'un tableau dans le prototype de la fonction. Cette taille est calculée par le compilateur. Il peut être inclus si vous le souhaitez, mais il sera ignoré lors de la compilation du code.

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

void UneFonction(double u[], double A[][10], double B[10][10]);

int main(int argc, char* argv[])
{
    double u[5], A[10][10], B[10][10];

    UneFonction(u, A, B);

    // Ceci affichera les valeurs allouées dans la fonction "UneFonction"
    std::cout << u[2] << "\n";
    std::cout << A[2][3] << "\n";
    std::cout << B[3][3] << "\n";

    return 0;
}

void UneFonction(double u[], double A[][10], double B[10][10])
{
    u[2] = 1.0;
    A[2][3] = 4.0;
    B[3][3] = -90.6;
}
```

Les tableaux dont la taille a été allouée dynamiquement peuvent également être envoyés à une fonction. Un exemple de code pour cela est présenté ci-dessous :

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

void UneFonction(double* u, double** A);

int main(int argc, char* argv[])
{
    double* u = new double [10];
    double** A = new double* [10];
    for (int i=0; i<10; i++)
    {
        A[i] = new double [10];
    }

    UneFonction(u, A);

    // Ceci imprimera les valeurs allouées dans la fonction "UneFonction"
    std::cout << u[2] << "\n";
    std::cout << A[2][3] << "\n";

    delete[] u;
    for (int i=0; i<10; i++)
    {
        delete[] A[i];
```

```
25      }
26      delete[] A;
27
28      return 0;
29  }
30
31  void UneFonction(double* u, double** A)
32  {
33      u[2] = 1.0;
34      A[2][3] = 4.0;
35  }
```

Exemple 3.5: Fonction pour calculer le produit scalaire de deux Vecteurs

On veut calculer le produit scalaire de deux vecteurs de nombres réels de longueur n. Le calcul du produit scalaire peut être intégré dans une fonction qui saisit les deux tableaux et la longueur n des deux vecteurs, et renvoie une variable réelle qui représente le produit scalaire des deux vecteurs. On aurait d'abord besoin d'allouer de la mémoire pour les deux vecteurs. On pourrait alors appeler la fonction qui calcule le produit scalaire, avant de finalement supprimer la mémoire allouée aux deux vecteurs. Le code correspondant est indiqué ci-dessous :

```
1   #include <iostream>
2
3   double CalculDuProduitScalaire(int size, double* a, double* b);
4
5   int main(int argc, char* argv[])
6   {
7       int n = 3;
8       double* x = new double [n];
9       double* y = new double [n];
10      x[0] = 1.0; x[1] = 4.0; x[2] = -7.0;
11      y[0] = 4.4; y[1] = 4.3; y[2] = 76.7;
12
13      double produit_scalaire = CalculDuProduitScalaire(n, x, y);
14
15      std::cout << "Le produit scalaire = " << produit_scalaire << "\n";
16      delete[] x;
17      delete[] y;
18
19      return 0;
20  }
21
22  double CalculDuProduitScalaire(int size, double* a, double* b)
23  {
24      double produit_scalaire = 0.0;
25      for (int i=0; i<size; i++)
26      {
27          produit_scalaire += a[i]*b[i];
28      }
29      return produit_scalaire;
30  }
```

3.6 Surcharge de fonction

On veut écrire une fonction pour multiplier un vecteur par un scalaire, et une autre fonction pour multiplier une matrice par un scalaire. Il semblerait naturel d'appeler ces deux fonctions `Multiplier`. Ceci est autorisé en C++ : on écrit différents prototypes de fonctions et fonctions pour ces deux opérations : le compilateur choisit alors la fonction correcte en fonction des arguments d'entrée. Ceci est donné dans le code ci-dessous et est connu sous le nom de surcharge de fonction.

```
#include <iostream>

void Multiplier(double scalaire, double* u, double* v, int n);

void Multiplier(double scalaire, double** A, double** B, int n);

int main(int argc, char* argv[])
{
    int n = 2;
    double* u = new double [n];
    double* v = new double [n];
    double** A = new double* [n];
    double** B = new double* [n];
    for (int i=0; i<n; i++)
    {
        A[i] = new double [n];
        B[i] = new double [n];
    }
    u[0] = -8.7; u[1] = 3.2;
    A[0][0] = 2.3; A[0][1] = -7.6;
    A[1][0] = 1.3; A[1][1] = 45.3;
    double s = 2.3, t = 4.8;

    // multiplication de vecteurs
    Multiplier(s, u, v, n);

    // multiplication de matrices
    Multiplier(t, A, B, n);

    delete[] u;
    delete[] v;
    for (int i=0; i<n; i++)
    {
        delete[] A[i];
        delete[] B[i];
    }
    delete[] A;
    delete[] B;

    return 0;
}

void Multiplier(double scalaire, double* u, double* v, int n)
{   // v = scalaire*u (un scalaire par un vecteur)
    for (int i=0; i<n; i++)
    {
        v[i] = scalaire*u[i];
    }
}

void Multiplier(double scalaire, double** A, double** B, int n)
{   // B = scalaire*A (un scalaire par une matrice)
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<n; j++)
        {
            B[i][j] = scalaire*A[i][j];
        }
    }
}
```

A noter qu'on peut surcharger des fonctions basées uniquement sur le nombre et le type des arguments et non sur le type de retour. Cela signifie qu'on ne peut pas avoir la fonction `bool Multiplier(double scalaire, double* u, double* v, int n)` à côté de la version qui a un type de retour `void`. En effet, le compilateur peut déduire la version correcte d'une fonction surchargée à partir des types de ses arguments et du contexte dans lequel elle est utilisée. Ce n'est pas le cas avec le type de retour, où vous pouvez appeler une fonction qui retourne quelque chose, mais ensuite convertir sa sortie en un autre type de retour, ou ignorer complètement sa sortie.

3.7 Déclarer des fonctions sans prototypes

Il est recommandé de fournir des prototypes de fonction avant d'écrire leur implémentation. C'est ainsi que la fonction `main`, ou toute autre fonction, reconnaîtra le nom et les types d'arguments de la nouvelle fonction. Cependant, il est possible de sauter l'écriture du prototype de la fonction en écrivant l'implémentation de la fonction avant sa première utilisation, comme le montre le code ci-dessous :

```
#include <iostream>

double Square(double x)
{
    return x*x;
}

int main(int argc, char* argv[])
{
    std::cout << "Le carré de 2 = " << Square(2) << "\n";
    return 0;
}
```

Si les prototypes ne sont pas donnés, alors les implémentations de fonctions doivent être ordonnées de manière à ce que chaque implémentation soit vue par le compilateur avant sa première utilisation. A noter que si deux fonctions sont mutuellement récursives, c'est-à-dire que les deux fonctions appellent l'autre fonction, alors il ne sera pas possible d'ordonner les fonctions de cette manière et donc des prototypes doivent être déclarés dans ce cas.

3.8 Fonctions récursives

Dans certaines applications, on peut appeler une fonction à partir de la même fonction : c'est ce qu'on appelle la récursivité, et c'est possible en C++. Une application simple de ceci est le calcul de la factorielle d'un entier positif `n`, noté `fact(n)`, et écrit mathématiquement $n!$, qui est défini par :

$$\begin{aligned} fact(n) &= n \times fact(n-1), & n > 1 \\ fact(n) &= 1, & n = 1 \end{aligned}$$

Le code pour implémenter cette définition récursive de la fonction factorielle est donné ci-dessous : on appelle simplement la fonction `CalculFacto` à partir de la même fonction autant de fois que nécessaire.

```
#include <iostream>
#include <cassert>

int CalculFacto(int n);

int main(int argc, char* argv[])
{
    int n = 7;
    std::cout << "La factorielle de " << n << " est " << CalculFacto(n) << "\n";

    return 0;
}

int CalculFacto(int n)
{
    assert (n > 0);
    if (n==1)
    {
        return 1;
    }
```

```
21      else    // n>1
22      {
23          return n*CalculFacto(n-1);
24      }
25  }
```

3.9 Utilisation des variables statiques

Une variable `int` statique reste en mémoire pendant que le programme est en cours d'exécution. Une variable normale ou automatique est détruite lorsqu'un appel de fonction où la variable a été déclarée est terminé.
Par exemple, on peut utiliser `static int` pour compter le nombre de fois où une fonction est appelée, mais une variable automatique ne peut pas être utilisée à cette fin.

```
1  #include <iostream>
2  using namespace std;
3
4  int fonction_test(){
5      static int cpt=0;
6      cpt++;
7      return cpt;
8  }
9  int main(void)
10 {
11     cout << fonction_test() << endl;
12     cout << fonction_test() << endl;
13     return 0;
14 }
```

Après exécution, la console affiche :

```
1
2
```

```
1  #include <iostream>
2  using namespace std;
3
4  int fonction_test(){
5      int cpt=0;
6      cpt++;
7      return cpt;
8  }
9  int main(void)
10 {
11     cout << fonction_test() << endl;
12     cout << fonction_test() << endl;
13     return 0;
14 }
```

Après exécution, la console affiche :

```
1
1
```

4. Documentation des codes

Au fur et à mesure qu'on commence à écrire plus de programmes, il y a souvent une tentation de "se contenter de coder" sans prêter une attention particulière à la qualité. Après tout « on sait généralement où on va et comprend le programme qu'on écrit ». Il est important de garder à l'esprit, cependant, que le code ne sera pas toujours aussi bien compris qu'il l'est maintenant. On peut revenir sur un dossier donné dans deux ans, car on a besoin de le corriger ou d'y ajouter de nouvelles fonctionnalités. Alternativement, on peut, à un moment donné, confier les programmes à quelqu'un d'autre qui a pour tâche de déterminer ce qu'on faisait.
Les programmes informatiques doivent être lisibles par l'homme, ainsi que par la machine. Même la plus petite partie du code peut s'avérer opaque à moins qu'on incluait suffisamment de commentaires pour aider le lecteur humain. On prend par exemple la fonction donnée ci-dessous, qui calcule la norme `p` d'un vecteur. Sans commentaires dans le code, il ne serait pas évident de savoir ce qui se passe, même s'il n'y a que quelques lignes de code. Un indice est donné dans le nom de la fonction, `CalculNorme`, mais à quoi sert-elle ? Quelle est la signification des arguments `s` et `p` ?

```
1  #include <cmath>
2  double CalculNorme(double* x, int s, int p)
3  {
4      double a = 0.0;
5      for (int i=0; i<s; i++)
6      {
7          double temp = fabs(x[i]);
8          a += pow(temp, p);
9      }
10     return pow(a, 1.0/p);
11 }
```

Dans le code ci-dessous, on donne une description de la fonction juste avant sa définition. Cette description donne, à la ligne 3, un moyen de faire correspondre les mathématiques de la fonction à son implémentation. Le reste de la description donne un autre endroit pour trouver plus d'informations sur la p-norme (lignes 4-6) et une explication de certains des arguments si nécessaire. Dans le corps de la fonction, la boucle a été commentée pour décrire son objectif fonctionnel : il s'agit de calculer une somme sur les éléments du vecteur. Enfin, la valeur de retour est commentée avec quelques mots d'explication.

```
1  #include <cmath>
2  /* Fonction pour calculer la p-norme d'un vecteur :
3   *  p-norme = [ Somme_i ( |x_i|**p ) ] **(1/p)
4   *  Voir "Une introduction à l'analyse numérique" sur internet
5   *      ou à la bibliothèque, pour la définition
6   *      de la p-norme d'un vecteur
7   *  x est un pointeur vers le vecteur de taille vecSize */
8
9  double CalculNorme(double* x, int vecSize, int p)
10 {
11     double a = 0.0;
12     // Boucle sur les éléments x_i de x, incrémentant la somme de |x_i|**p
13     for (int i=0; i<s; i++)
14     {
15         double temp = fabs(x[i]);
16         a += pow(temp, p);
17     }
18     // Renvoie la p-ième racine  de la somme
19     return pow(a, 1.0/p);
20 }
```

La documentation du code est parfois plus un art qu'une science. Il y a un équilibre à trouver concernant le bon niveau de documentation. Trop de commentaires peuvent rendre le programme moins lisible plutôt que plus lisible. Il est conseillé de décrire quelle partie du problème le code résout et, peut-être, comment il résout ce problème. Il ne faut pas décrire le code avec trop de détails. Par exemple, le commentaire sur la boucle de la ligne 12 du code ci-dessus aurait pu lire `// Boucle sur les valeurs de i allant de 0 à vecSize-1`
Bien que ce commentaire soit précis (décrivant la plage de la variable de boucle), il ne fait rien pour aider un programmeur à comprendre le code.
Le formatage de la documentation du code peut également aider à la lisibilité. Une astuce simple est que l'utilisation

de lignes vides pour diviser le code et les commentaires en sections peut rendre le code plus lisible. Si on souhaite mettre l'accent sur quelque chose, on peut simuler le soulignement avec des traits d'union ou des traits de soulignement, par exemple.

```
// Commentaire très important
// --------------------------
```

Alternativement, on peut souligner quelque chose en le mettant dans une boîte :

```
/**************************************************
 **************************************************
 **                CalculNorme(...)                **
 **                                                **
 **   Fonction pour calculer la norme p du vecteur **
 **************************************************
 **************************************************/
```

5. Exercices

Exercice 1 :

Ecrire une fonction de prototype int puissance(int a, int b) qui calcule a^b, a et b sont des entiers (cette fonction n'existe pas en bibliothèque). La mettre en oeuvre dans le programme principal.

Exercice 2 :

tab1 et tab2 sont des variables locales au programme principal.

```
int tab1[20] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
int tab2[20] = {-19,18,-17,16,-15,14,-13,12,-11,10,-9,8,-7,6,-5,4,-3,2,-1,0};
```

Écrire une fonction de prototype void affiche(int *tx, in taille) qui permet d'afficher les nombres suivant un tableau par lignes de 5 éléments.
La mettre en œuvre dans le programme principal pour afficher tab1 et tab2.

Exercice 3 :

`liste` est une variable locale au programme principal.

```
float liste[8] = {1.6,-6,9.67,5.90,345.0,-23.6,78,34.6};
```

Ecrire une fonction de prototype float min(float *tx, int taille) qui renvoie le minimum de la liste.
Ecrire une fonction de prototype float max(float *tx, int taille) qui renvoie le maximum de la liste.
Les mettre en œuvre dans le programme principal.

Exercice 4 :

Saisir les 3 couleurs d'une résistance, afficher sa valeur. Une fonction de prototype float conversion(char *couleur) calcule le nombre associé à chaque couleur. "couleur" est une chaîne de caractères.

Exercice 5 :

Écrire un programme pour calculer et afficher les racines de $ax^2+bx+c=0$. Le programme doit contenir :

- Une fonction de prototype void saisie(float &aa, float &bb, float &cc) permet de saisir a, b, c.
  Ici, le passage par référence est obligatoire puisque la fonction " saisie " modifie les valeurs des arguments
- Une fonction de prototype void calcul(float aa, float bb, float cc) exécute les calculs et affiche les résultats (passage par référence inutile).
- a, b, c sont des variables locales au programme principal
- Le programme principal se contente d'appeler saisie(a,b,c) et calcul(a,b,c).

Exercice 6 : Calculs scientifiques

On sait qu'une résistance R, traversée par un courant d'intensité I, s'échauffe (effet Joule).
La loi de variation de la résistance avec la température est donnée par la relation approchée :

$$R_t = R_0 \times \big(1 + (\alpha \times t)\big)$$

où

- R_0 est la résistance à 0°C.
- α est le coefficient de température du matériau utilisé.
- T est la température en ° Celsius.

Écrire un programme de type menu qui permet de calculer l'un des 4 paramètres en fonction des 3 autres.
Le choix s'effectue suivant le principe suivant :

Quel paramètre voulez-vous déterminer ?

1. Résistance R_t
2. Résistance R_0
3. Coefficient de température α
4. Température T
5. Quitter le programme

Votre choix : ?

On prévoira, bien sûr, pour chaque cas, la lecture des trois paramètres donnés et l'affichage du résultat obtenu (quatrième paramètre) avec un format correct. Vous utiliserez le jeu d'essais suivant :

Test	Données à saisir			Paramètre à calculer
Test n°1	$R_0 = 100\ \Omega$	$\alpha = 3.9 \times 10^{-3}$	$T = 128°C$	R_t
Test n°2	$R_0 = 100\ \Omega$	$R_t = 123.64\ \Omega$	$T = 270°C$	α
Test n°3	$R_0 = 100\ \Omega$	$R_t = 161.7\ \Omega$	$\alpha = 3.9 \times 10^{-3}$	T

Table 1. Cahier des charges