

Programmation C++

Passage du C au C++, fonctionnalités du C++

Pr Y. ES-SAADY
y.essaady@uiz.ac.ma

EST Agadir

2018/2019

Objectifs de ce cours

- **Présenter quelques nouveautés du C++ par rapport au C, mais sans encore faire vraiment de programmation orientée objet:**
 - les entrées/ sorties,
 - types et declarations,
 - allocation dynamique,
 - surcharge de fonctions,
 - arguments par défaut ,
 - transmission par référence,
 - patron de fonctions,
 - les fonctions en ligne,
 - rappels sur les structures et tableau de structures

Le langage C++

- Créé en 1980 par Bjarne Stroustrup
 - Standardisation de C++ en 1998
 - Version corrigée de C++ en 2003
 - Nouvelle norme pour le langage C++ en septembre 2011
- Extension du langage C, pour la programmation objet.
- Caractéristiques de C++
 - Langage compilé
 - Langage procédural (procédures, fonctions, méthodes...)
 - Programmation orienté objet (classes)
 - Programmation générique (templates)
 - Efficace et souple
 - Programmation fine du matériel.
 - Portable (de nombreux environnement/compilateurs disponibles)

Spécificités de C++ sur langage de C

- C++ dispose d'un certain nombre de spécificités par rapport a C qui ne sont pas axées sur l'orienté objet :
 - le commentaire
 - l'emplacement libre des déclarations
 - passage de paramètres par référence
 - la surdéfinition de fonction
 - les fonctions en ligne
 - les opérateurs new et delete

Exemple 1

Exemple de fichier main.cpp

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    std::cout << "Bonjour !\n";
```

```
    return 0;
```

```
}
```

Fichier en
tête

Fonction
principale
Entrée du
programme

Début
bloc

Instruction

Fin bloc

Exemple 2

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, c;
    cout << "Entrez deux chiffres : ";
    cin >> a;
    cin >> b;
    c=a+b;
    cout << "c = " << c;
    cout << "\n Fin de traitement...\n\n";
    return 0;
}
```

Déclaration de
variables

Saisie de
la valeur
de a

Les commentaires (lisibilité !)

Syntaxe en C et en C++

```
/* Ceci est un commentaire sur plusieurs lignes  
...  
Fin du commentaire */
```

Syntaxe en C++ seulement

```
// ceci est un commentaire sur une ligne
```

```
int i = 0; /* initialisation de i */  
i = 2; // affectation de la valeur 2 à la variable i  
cout << "i=" << i << endl;
```

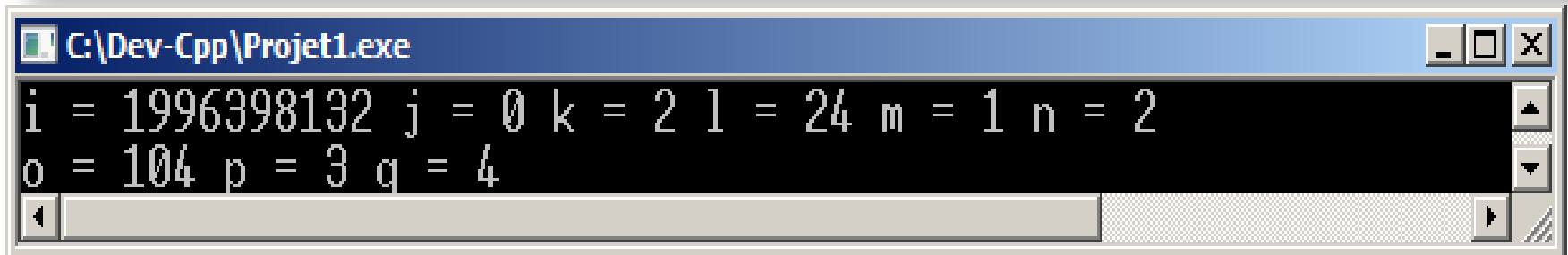
Types de base

- **Les entiers relatifs** (1,2,-3 etc...) : **int** (2 octet).
(on peut leur ajouter les attribut : **short**, **long**, **signed**, **unsigned**)
- **Les réels ou flottants** (0.12, PI, ...) : **float** (4 octet).
- **Les réels double précision** : **double** (8 octets).
(on peut lui ajouter l'attribut : **long**)
- **Les caractères** ('a', 'b', 'c', '#', '3' ...) : **char** (1octet)
(on peut lui ajouter les attributs **signed** ou **unsigned**)
- **Les types booléens** : **bool** (Il peut valoir true ou false)
 - Pas de type booléen en C.
 - En C on utilise le type int
 - **0** représente faux.
 - Toute valeur **non nulle** représente vrai.
 - En C++, le type booleen est **bool**

Les déclarations

- Peuvent être suivies d'une affectation.

```
int i; int j=0; int k, l;  
int m=1, n=2;  
int o, p=3, q=4;  
cout << "i = " << i << " j = " << j << " k = " << k;  
cout << " l = " << l << " m = " << m << " n = " << n << endl;  
cout << "o = " << o << " p = " << p << " q = " << q << endl;
```



```
C:\Dev-Cpp\Projet1.exe  
i = 1996398132 j = 0 k = 2 l = 24 m = 1 n = 2  
o = 104 p = 3 q = 4
```

Les E/S standards

- Il est toujours possible d'utiliser les fonctions d'E/S déclarées dans **<stdio.h>**
- Le C++ introduit de nouvelles facilités
 - grâce aux opérateurs d'insertion dans un flux de sortie et d'extraction dans un flux d'entrée
 - traitement uniforme pour les types primitifs (plus besoin de préciser le format de conversion)
 - il existe 3 flux prédéfinis qui correspondent au clavier (**cin**), à l'écran (**cout**) et le périphérique standard d'erreur (**cerr**)

Les E/S standards

- Pour utiliser ces flux il faut inclure la bibliothèque `<iostream>`
 - le flux standard de sortie est `cout` et l'opérateur associé est `<<`
 - le flux standard d'entrée est `cin` et l'opérateur associé est `>>`
 - le flux standard des erreurs est `cerr` et l'opérateur associé est `<<`

```
cout << "saisir la variable x : ";  
cin >> x;  
cout << endl;  
cout << "la valeur de x est : " << x << endl;
```

Rappel sur les pointeurs et références

- `int x=3; int * p ;` // un pointeur sur un entier
- `p = &x ;` // p vaut l'adresse de x
- `*p` désigne la valeur contenue dans l'emplacement mémoire dont l'adresse est p.
- `int y = *p ;` // y vaut 3
- On peut aussi déclarer une référence à une variable existante.
- `int & z = x ;` // z est une référence a x, z vaut 3
- Si on change la valeur de x, la valeur de z est aussi changée et inversement. Idem avec *p.
 - `x = 4;` // x et z valent 4, *p aussi
 - `z = 5;` // x et z valent 5, *p aussi
 - `*p = 6;` // *p, x et z valent 6
- Noter que les opérateurs **&** et ***** sont inverses l'un de l'autre. On a toujours : **`*(&x) = x`** et **`&(*p) = p`**

Opérateurs new et delete

- En C, la gestion dynamique de la mémoire fait appel aux fonctions **malloc** et **free**.

- En C++, on utilise les fonctions **new** et **delete**

Avec la déclaration : **int * a ;**

en C++ on alloue dynamiquement comme cela : **a = new int ;**

- alors qu'en C il fallait faire comme ceci :

a= (int *) malloc (sizeof(int)) ;

- Plus généralement, si on a un type donné type, on alloue une variable avec : **new type ;**
- ou un tableau de n variables avec : **new type [n] ;**

Opérateurs new et delete

■ Exemple

```
double x = 8.0;  
double *p, *q;  
p = &x;  
q = new double;  
*q = 4.0;  
cout << " *p = " << *p << endl;  
cout << " *q = " << *q << endl;
```

Résultat

*p = 8

*q = 4

Opérateurs new et delete

- La mémoire allouée dynamiquement doit être desallouée via l'opérateur **delete** comme cela : **delete a;**
- Plus généralement, on désalloue une variable x (**allouée avec new**) avec : **delete x;**
- ou un tableau de variables (**allouée avec new []**) avec : **delete [] x ;**

```
double * p = new double ; // allouer la memoire
// utilisation de p et *p...
delete p ;                  // desallouer la memoire
```

■ Remarque

En C++, bien que l'utilisation de malloc et free soit toujours permise, il est très conseillé de n'utiliser que new et delete.

Opérateurs new et delete

- Il est important de ne pas oublier [] lors de désallocation du tableau.
- **Exemple**

```
int n = 9;  
int * t = new int[n];  
for( int i=0; i<n; i++ ) {  
    t[i] = i;  
    cout << t[i];  
}  
delete [] t;
```

Résultat

012345678

Tableau dynamique de tableaux dynamiques

- Voici un exemple d'allocation d'un tableau dynamique à deux dimensions (i.e. un tableau dynamique à une dimension de tableaux dynamiques à une dimension).

```
int N = 3, i, j;
int** A = new int*[N];
for( i=0; i<N; i++ ){
    A[i] = new int[N];
    for( j=0; j<N; j++ )
        A[i][j]=(i+j) % 2;
}
for( i=0; i<N; i++ ) // Restitution de la memoire allouée : on
    delete [] A[i];  // commence par desallouer les sous tableaux puis
delete [] A;         //on desalloue le tableau principal
```

Les fonctions

Syntaxe des fonctions

```
typeRetour nomDeLaFonction ( spécification des  
                                paramètres formels )  
{  
    suite de déclarations de variables locales et  
    d'instructions  
}
```

- *TypeAuRetour* indique le **type** du résultat de la fonction.
- Les paramètres formels (ou arguments) doivent être **séparés par des virgules**, et sont **typés**.

Les fonctions (Exemple)

```
#include <iostream>
using namespace std;
int max(int a, int b)
{
    int m;
    if (a<b) {
        m=b;
    } else {
        m=a;
    }
    return m;
}
int main()
{
    int x, y, z ;
    cout << " donnez la valeur de x :" ;
    cin >> x;
    cout << " donnez la valeur de y :" ;
    cin >> y;
    z=max(x,y); // appel de la fonction max
    cout <<"le plus grand des deux est:"<< z << endl;
    return 0;
}
```

Prototype de fonction

- une fonction peut être appelée avant qu'elle soit définie . Dans ce cas, il faut au préalable faire une déclaration de cette fonction en utilisant un prototype de fonction.

Syntaxe

TypeRetour **nom_Fonction** (*types des paramètres*);

Prototype de fonction

Exemple

```
int min(int , int ) ; //déclaration de la fonction (prototype)  
                        //ici, les noms des paramètres sont optionnels  
  
int main()           // programme principale  
{ int i, j, m;  
    .....  
    m = min(i, j); // appel de la fonction  
    .....  
}  
  
int min(int x, int y) //définition complète de la fonction  
{  
    return (x < y)? x : y;  
}
```

Compilation séparée

- Il est possible de définir le corps d'une fonction dans un fichier séparé. Voici le contenu du fichier **main.cpp**:

```
int max( int, int );  
int main()  
{  
    return max(0,1);  
}
```

- Seul le prototype de la fonction max a été défini dans **main.cpp**. On peut implémenter le corps de la fonction max dans un fichier séparé.

Surcharge de fonction (C++)

- On parle de **surcharge** (surdéfinition) lorsqu'un symbole prend plusieurs significations différentes.
- On peut utiliser un même nom de fonction dès lors que les arguments sont différents (profil de fonction), par exemple:

```
int max( int a, int b ) {  
    if (a>b) return a; return b;  
}  
  
double max ( double a, double b ) {  
    if (a>b) return a; return b;  
}  
  
int max( int a, int b, int c ) {  
    return max(a, max(b,c));  
}
```

Surcharge de fonction (C++)

■ Exemples d'appel de la fonction max:

```
void main() {  
    cout << "max(1,2) = " << max(1,2) << endl;  
    cout << "max(1,2,3) = " << max(1,2,3) << endl;  
    cout << "max(3.4,7.2) = " << max(3.4,7.2);  
}
```

Résultat

max(1,2) = 2

max(1,2,3) = 3

max(3.4,7.2) = 7.2

Arguments par défaut (C++)

- En C il est indispensable que l'appel de la fonction contienne exactement le même nombre et type d'arguments que dans la déclaration de la fonction.
- C++ permet de s'affranchir de cette contrainte en permettant l'usage **d'arguments par défaut**.
- Les derniers paramètres d'une fonction C++ peuvent être munis d'une valeur par défaut : si ces paramètres sont absents lors de l'appel de la fonction, c'est leur valeur par défaut qui est prise en compte

```
int somme( int a, int b, int c = 0, int d = 1 )  
{  
    return a+b+c+d;  
}
```

Arguments par défaut (C++)

■ Exemples d'appel de la fonction somme:

```
void main() {  
    int x = 1, y = 2, z = 3, t = 4;  
    cout << "somme(x,y) = " << somme(x,y) << endl;  
    cout << " somme(x,y,z) = " << somme(x,y,z) << endl;  
    cout << " somme(x,y,z,t) = " << somme(x,y,z,t);  
}
```

Les paramètres par défaut sont obligatoirement les derniers de la liste.

Ils ne sont déclarés que dans le prototype de la fonction et pas dans sa définition.

Résultat

somme(x,y) = 4
somme(x,y,z) = 7
somme(x,y,z,t) = 10

Rappel sur les macros en C

- En C, on peut définir une macro avec le mot-clé **define** :

```
#define carre(A) A*A
```

- Cela permet à priori d'utiliser **carre** comme une fonction normale :

```
int a = 2 ;  
int b = carre(a) ;  
cout << "a = " << a << " b = " << b << endl ;
```

Résultat

a = 2 b = 4

- Quand il voit une macro, le C remplace partout dans le code de l'expression **carre(x)** par **x*x**.
- L'avantage d'une macro par rapport à une fonction est la rapidité d'exécution

Rappel sur les macros en C

■ Inconvénients d'une macro

- espace mémoire du programme plus grand
- les effets de bord des macros ???

■ Si l'on programme :

```
int a = 2 ;  
int b = carre (a++) ;  
cout << "a = " << a << " b = " << b << endl ;
```

■ On attendrait le résultat suivant : **a = 3 b = 4**

■ En réalité, ce n'est pas le cas car, à la compilation, le C remplace malheureusement **carre(a++)** par **a++*a++**. Et à l'exécution, le programme donne le résultat suivant : **a = 4 b = 4**

Fonctions inline (C++)

- Le C++ palie à l'inconvénient des macro en permettant de définir des fonctions, dites **en ligne**, avec le mot clé '**inline**'.

```
inline int carre( int x )  
{  
    return x*x;  
}
```

- Une fonction définie avec le mot-clé **inline** aura le même avantage qu'une macro en C, à savoir un gain de temps d'exécution et le même inconvénient, à savoir une plus grande place mémoire occupée par le programme.
- L'avantage des fonctions en ligne est la disparition des effets de bord.

Fonctions inline (C++)

■ Exemple:

```
#include <iostream>
using namespace std;
inline int carre( int x )
{
    return x*x;
}

int main(int argc, char *argv[])
{
    int a = 2 ;
    int b = carre(a++) ;
    cout << " a = " << a << " b = " << b << endl ;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```



Passage de paramètres par valeur et passage par référence

■ Deux de types de passage de paramètres

- Passage de paramètres par valeur
- Passage de paramètres par référence

- **En C**, les arguments sont passés par valeur. Ce qui signifie que les paramètres d'une fonction C sont toujours en entrée de la fonction et pas en sortie de la fonction ;
- Autrement dit les paramètres d'une fonction ne peuvent pas être modifiés par la fonction.

Passage de paramètres par valeur

```
void echange(int a, int b) {  
    int c = a ;  
    a = b ;  
    b = c ;  
}  
main() {  
    int n = 10 ; int p = 20 ;  
    cout << " avant appel : n=" << n << " p=" << p << endl ;  
    echange(n, p) ;  
    cout << " après appel: n=" << n << " p=" << p << endl ;  
}
```

Résultat

avant appel : n=10 p=20

après appel : n=10 p=20

Passage de paramètres par référence: en C++

- Il suffit de mettre le symbole **&** après le type de la variable

```
void echange(int &a, int &b) {  
    int c = a ;  
    a = b ;  
    b = c ;  
}  
main() {  
    int n = 10 ; int p = 20 ;  
    cout << " avant appel : n=" << n << " p=" << p << endl ;  
    echange(n, p) ;  
    cout << " après appel: n=" << n << " p=" << p << endl ;  
}
```

Résultat

avant appel : n=10 p=20

après appel : n=20 p=10

Passage de paramètres par valeur et passage par référence

Passage par valeur	Passage par référence
<ul style="list-style-type: none">▪ <code>int x;</code>▪ Le paramètre formel <code>x</code> est une variable locale▪ <code>x</code> est une copie du paramètre effectif▪ Ne peut changer le paramètre effectif▪ Le paramètre effectif peut être une variable, une constante ou une expression▪ Le paramètre effectif est en lecture seule	<ul style="list-style-type: none">▪ <code>int& x;</code>▪ Le paramètre formel <code>x</code> est une référence locale▪ <code>x</code> est un synonyme du paramètre effectif▪ Peut changer le paramètre effectif▪ Le paramètre effectif ne peut être qu'une variable▪ Le paramètre effectif est en lecture écriture

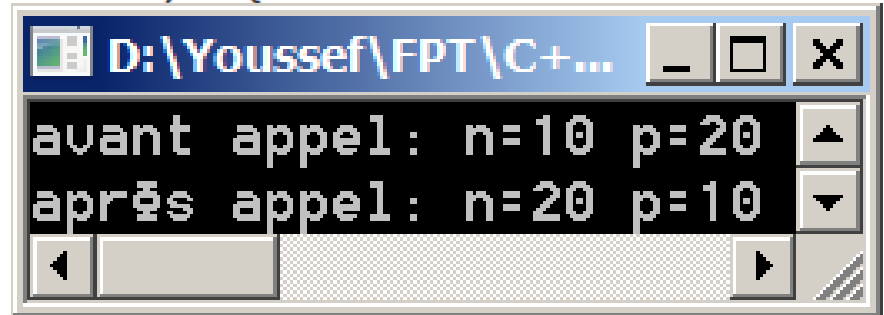
Passage de paramètres: Exercice

- Ecrire une fonction permettant d'échanger les contenus de 2 variables de type int fournies en argument
 - en transmettant l'adresse des variables concernées (seule méthode utilisable en C),
 - en utilisant la transmission par référence.
- Dans les deux cas, on écrira un petit programme d'essai (main) de la fonction.

Passage de paramètres: correction de l'exercice

- En transmettant l'adresse des variables concernées (seule méthode utilisable en C)

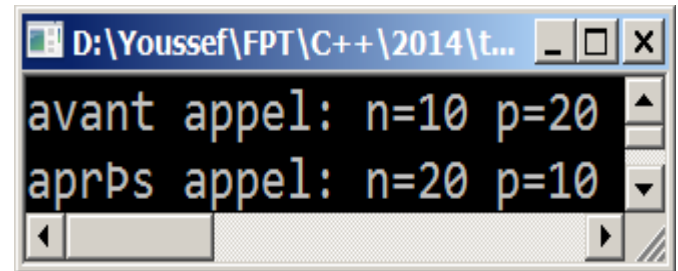
```
#include <cstdlib>
#include <iostream>
using namespace std;
void exchange(int* a, int* b) {
    int c = *a ;
    *a = *b ;
    *b = c ;
}
main() {
    int n = 10 ; int p = 20 ;
    cout <<"avant appel: n="<<n<<" p="<<p<<endl ;
    exchange(&n, &p) ;
    cout <<"après appel: n="<<n<<" p="<<p<<endl ;
    system("PAUSE") ;
}
```



Passage de paramètres: correction de l'exercice

■ En utilisant la transmission par référence.

```
#include <cstdlib>
#include <iostream>
using namespace std;
void echange(int& a, int& b) {
    int c = a ;
    a = b ;
    b = c ;
}
main() {
    int n = 10 ; int p = 20 ;
    cout <<"avant appel: n="<<n<<" p="<<p<<endl;
    echange(n, p) ;
    cout <<"après appel: n="<<n<<" p="<<p<<endl;
    system("PAUSE");
}
```



Modèle de fonctions

- (fonction générique ou fonction modèle ou fonction template)
- Lorsque l'algorithme est le même pour plusieurs types de données, il est possible de créer un patron de fonction.
- C'est un modèle à partir duquel le compilateur générera les fonctions qui lui seront nécessaires.
- Exemple: Faire des fonctions de max de 2 variables de même type.
- Sans Modèle de fonctions : Il faudrait surcharger la fonction **Max** avec autant de fonctions **Max** que de types de données sur lesquels appliquer la fonction Max.

```
int  Max(int a,int b) { return a>b?a:b;}  
float  Max (float a,float b) { return a>b?a:b;}  
double  Max (double a,double b) { return a>b?a:b;}
```

Modèle de fonctions

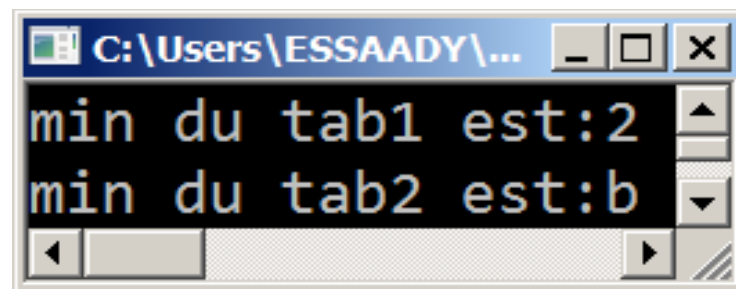
■ Avec les fonctions templates :

```
template <class T> T max ( T a, T b)  
{ return a>b ? a : b; }
```

- A chaque fois que le compilateur C++ rencontrera un appel de la fonction max pour des données de type double, il engendrera le code de la fonction max pour le type double.

Modèle de fonctions: Exemple

```
#include <cstdlib>
#include <iostream>
using namespace std;
template <class T> T minTab( T a [], int taille)
{ T min= a[0];
  for (int i=1; i<taille; ++i)
    if(a[i]<min) min=a[i];
  return min;
}
main()
{ int tab1[]={10,4,8,2};
  int m=minTab(tab1,4);
  cout <<"min du tab1 est:" <<m<<endl;
  char tab2[]={'e','x','b','c','y'};
  char c=minTab(tab2,5);
  cout << "min du tab2 est:" <<c<<endl;
  system("PAUSE");
}
```



Rappel sur les structures

- Une **structure** est un type composé d'une séquence de membres de types divers, portants des noms : **les champs**.

```
struct Tdate {  
    int annee;  
    int mois;  
    int jour;  
}
```

```
struct Client {  
    char nom[20] ;  
    char prenom[20] ;  
    unsigned age;  
};
```

Tableau de structures

- Il est bien entendu possible de regrouper sous un même nom, plusieurs structures de même type pour former un tableau de structure.
- **Exemple**: Déclaration d'un tableau de 20 clients

```

struct Client Clt[3];

Clt[0].age=20;

struct Client * C;

C= new struct Client [4];

C[0].age=20;

*(C+1).age=50;

(C+1)->age=50;

```

Clt

Karimi	Ait Brahimi	Hamidi
Ali	Ahmed	Fatima
20	50	25

C*

Karimi	Ait Brahimi		
Ali	Ahmed		
20	50		

Exemple

```
#include <cstdlib>
#include <iostream>
using namespace std;
struct employe{
    char nom[10];
    char prenom[10];
    double salaire;
};
main( ){
    struct employe E;
    cout<<"Entrez le nom :"; cin>>E.nom;
    cout<<"Entrez le prenom :"; cin>>E.prenom;
    cout<<"Entrez le salaire :"; cin>>E.salaire;
    cout<<E.nom<<" "<<E.prenom<<" "<<E.salaire<<endl;
    system("PAUSE");
}
```

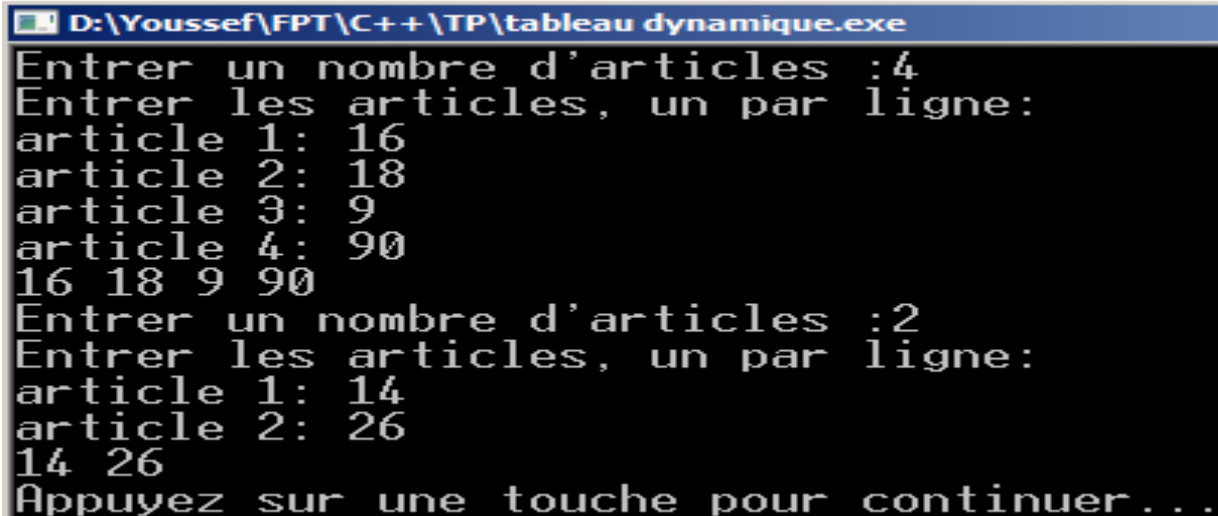
Exemple : utilisation de fonctions et tableaux dynamiques

```
#include <iostream>
using namespace std;
/* la fonction get() présentée ici crée un tableau dynamique*/
void get (double* & a, int& n)
{
    cout << "Entrer un nombre d'articles :";
    cin >> n;
    a= new double[n];
    cout << "Entrer les articles, un par ligne:\n";
    for (int i=0; i<n; i++){
        cout << "article "<< i+1 <<": ";
        cin >> a[i];
    }
}

void Afficher (double *a, int n) // fonction qui affiche le tableau
{
    for (int i=0;i<n;i++)
        cout << a[i] << " ";
    cout <<endl;
}
```

Exemple : utilisation de tableaux dynamiques

```
int main()           // programme principale
{
    double * A;      // A ici est un simple pointeur non encore alloué
    int n;
    get (A, n);
    Afficher (A, n); // A ici est un tableau de n double
    delete []A;
    get (A, n);
    Afficher (A, n); // A est de nouveau un tableau de n double
    system("PAUSE");
    return EXIT_SUCCESS;
}
```



```
D:\Youssef\FPT\C++\TP\tableau dynamique.exe
Entrer un nombre d'articles :4
Entrer les articles, un par ligne:
article 1: 16
article 2: 18
article 3: 9
article 4: 90
16 18 9 90
Entrer un nombre d'articles :2
Entrer les articles, un par ligne:
article 1: 14
article 2: 26
14 26
Appuyez sur une touche pour continuer...
```