Chapitre 3. Notions de base

1. Introduction

Les notions de base de la programmation en C restent valables en C++, néanmoins C et C++ diffèrent sur quelques conventions (déclaration des variables et des fonctions, nouveaux mots clés…)

2. Structure d'un programme en langage C++

2.1. Fonction main

Tout programme doit avoir un point d'entrée nommé main.

```
int  main()
{

return 0;
}
```

La fonction main est la fonction appelée par le système d'exploitation lors de l'exécution du programme

- **{ }** délimitent le corps de la fonction
- **main** retourne un entier au système: 0 (zéro) veut dire succès
- Chaque expression doit finir par**; (point-virgule)**

2.2. Commentaires

La façon d'introduire en langage C des commentaires multi-lignes, avec /* et */, continue à être permise. Mais on peut également utiliser deux signes de division //. Dans ce cas, tout ce qui suit sur la ligne est alors considère comme un commentaire. On parle de commentaire de fin de ligne.

/* Un commentaire en une seule ligne */

// Un commentaire jusqu'à la fin de cette ligne

```
/
#include <iostream>
using namespace std;
int main(void)
{
int n ;
int m;
cout << "je suis un etudiant de l'universite de khemis miliana \n Mon catre est de numero: " ;//Introduire votre université
cin >> n ;
cout << "mon age ";// introduire votre age
cin>>m ;
cout<< " Mon annee de naissance est " << 2022-m << '\n' ;// affiche l'année de naissance
}
```

2.3. Fichiers Sources

Un programme est généralement constitué de plusieurs modules, chaque module est composé de deux fichiers sources:

- Un fichier contenant la description de l'interface du module
- Un fichier contenant l'implémentation proprement dite du module

Un suffixe est utilisé pour déterminer le type de fichier :

- .h, .H, .hpp, .hxx : pour les fichiers de description d'interface (header files ou include files).
- .c, .cc, .cxx, .cpp, .c++ : pour les fichiers d'implémentation

Dans un fichier source on peut trouver:

- Commentaires
- Instructions pré-processeur
- Instructions C++

2.4. Bibliothèque de C++

Le C++ dispose d'une bibliothèque standard très riche, elle comporte un très grand nombre d'outils (fonctions, types, …) qui permettent de faciliter la programmation. Elle comporte notamment de nombreux patrons de classes et de fonctions permettant de mettre en œuvre les structures de données les plus importantes (vecteurs dynamiques, listes chaînées, chaînes...) et les algorithmes les plus usuels.

Le C++ dispose de la totalité de la bibliothèque standard du C, y compris de fonctions devenues inutiles ou redondantes. Afin d'utiliser, en C++, les outils qui existaient dans la bibliothèque standard de C (stdio.h, string.h, …) ainsi que certains nouveaux outils, il suffit de spécifier avec la directive include le fichier entête (.h) souhaité.

Exemple:

```
#include <iostream>  // En C++: <<Input Output Stream>> ou bien <<Flux d'entrée sortie>>
#include <cstdio>    // En C : « Flux d'entrée-sortie »
```

```
#include <iostream>
#include <cstdio>      // Les librairie C
using namespace std;
int main()
{     std ::cout << "essalam alaykom"<<std ::endl;
      printf(" wa 3alaykom esslam \n ") ;    // Les
return 0;
}
```

2.5. Espace de noms en C++

Pour des raisons liées à la POO (généricité, modularité…) et pour éviter certains conflits qui peuvent surgir entre les différents noms des outils utilisés (prédéfinis ou définis par l'utilisateur), C++ introduit la notion de namespace (espace de noms), ce qui permet de définir des zones de déclaration et de définitions des différents outils (variables, fonctions, types,…). Ainsi, chaque élément défini dans un programme ou dans une bibliothèque appartient désormais à un namespace. La plupart des outils d'Entrées / Sorties de la bibliothèque de C++ appartiennent à un namespace nommé "std".

Syntaxe:

```
using namespace std;
```

Exemple :

```
#include <iostream>
#include <conio.h>
using namespace std;  // Importation de l'espace de nom

 int main()
{
    double x, y;
    // Le préfixe n'est plus requis :
    cout << " la valeur de X est :";
    cin >> x;
    cout << "la valeur de Y est:";
   // Il est toujours possible d'utiliser le préfixe :
  std::cin >> y;
   cout << " x + y = " << x +y << endl;

}
```

Il est possible aussi de définir un espace de nom alors les identificateurs de cet espace seront préfixés.

Syntaxe :

```
namespace nom
{
       // Placer ici les déclarations faisant partie de l'espace de nom
}
```

Exemple

```
#include <iostream>
using namespace std;  // Importation de l'espace de nom

namespace DBKM
{
    void afficher_adresse()
     {    cout << "Université Djilali Bounaama de Khemis Miliana,";
          cout << "Route de Theniet El Had, 44225,"<<endl<< " Khemis Miliana,"<<endl ;
     }
     void afficher_telephone()
    {    afficher_adresse();   // Préfixe non requis. car dans le même espace de  nom :
         cout << "Tel :   027 55 68 34\n";
     }
}

int main()
{
    DBKM::afficher_telephone();
     return 0;
}
```

2.6. Les entrées/sorties en C++

On peut utiliser les routines d’E/S da la bibliothèque standard de C (<stdio.h>). Mais C++ possède aussi ses propres possibilités d’E/S.
Ces instructions d’E/S exigent un nouveau fichier en-tête, appelé **iostream.h** (pour flot d’entrée-sortie). cin et cout sont dans la bibliothèque iostream,
Syntaxe : La syntaxe d’une sortie est : **cout << expr1 << ...<< exprn ;**
Ou expr1, . . ., exprn sont des expressions de type caractère, entier, réel, chaine de caractères ou pointeur (on obtient l’adresse dans ce dernier cas).
La syntaxe d’une entrée est : **cin** >> var ;
Ou var est une variable de type caractère, entier, réel ou chaine de caractères.
On peut entrer plusieurs valeurs à la fois.
Exemple 1

```
#include <iostream> // pour utiliser cin et cout
using namespace std;
int main(void)
{
int n ; // n est un entier
cout << "l'age d'etudiant \n  Entrer un entier : " ;
cin >> n ; // saisie de n
cout << "l'annee de naissance est: " << 2022-n ; '\n' ;
}
```

Exemple 2: écrire un programme qui affiche "je suis un étudiant de l'université de khemis miliana", puis il demande d'entrer le numéro de votre carte d'étudiant et votre âge et afficher l’année de votre naissance.

```
#include <iostream>
using namespace std;
int main(void)
{
int n ;
int m;
cout << "je suis un etudiant de l'universite de khemis miliana \n Mon catre est de numero: " ;
cin >> n ;
cout << "mon age ";
cin>>m ;
cout<< " Mon annee de naissance est " << 2022-m << '\n' ;
}
```

L’expression endl permet le retour à la ligne.
Les avantages des nouvelles possibilités d'E/S :

- Vitesse d'exécution plus rapide.
- Il n'y plus de problème de types
- Autres avantages liés à la POO.

2.7. Bibliothèque iomanip

Il existe d’autres possibilités de modifier la façon dont les éléments sont lus ou écrits dans le flot:

- **dec**: Lecture/écriture d'un entier en décimal.
- **oct:** Lecture/écriture d'un entier en octal.
- **hex:** lecture/écriture d'un entier en hexadécimal.
- **endl:** Insère un saut de ligne.
- **setw(int n):** Affichage de n caractères.
- **setprecision(int n):** Affichage de la valeur avec n chiffres avec éventuellement un arrondi de la valeur.
- **setfill(char):** Définit le caractère de remplissage flush vide les tampons après écriture.

Exemple

```
#include <iostream>
#include <iomanip>        // attention a bien inclure cette librairie
using namespace std ;

int main()
{   int i=1348;
    float p=45.8765;
    cout << "|" << setw(8) << setfill('*') << hex << i << "|" <<endl;
              cout<< "|" << setw(7) << setfill('+') <<setprecision(4) << p <<"|"<< endl;
return 0;
}
```

3. Variables et constantes

3.1. Noms de variables

En C++, il y a quelques règles qui régissent les différents noms autorisés ou interdits :

- Les noms de variables sont constitués de lettres, de chiffres et du tiret-bas « _ » uniquement. Le double souligné "__" au début du nom est réservé aux variables/fonctions du système.
- Le premier caractère doit être une lettre (majuscule ou minuscule).
- On ne peut pas utiliser d'accents.
- On ne peut pas utiliser d'espaces dans le nom

Exemple:

ageEtudiant, nom_etudiant, NOMBRE_Etudiants **: Noms valides.**

Ageétudiant, nom etudiant: **Noms non valides**.

3.2. Types de base

Après l'identification du nom de la variable, L'ordinateur doit connaitre ce qu'il a dans cette variable, il faut donc indiquer quel type (nombre, mot, lettre, ou …) d'élément va contenir la variable que nous aimerions utiliser. Le tableau 1 présente la liste des types de variables que l'on peut utiliser en C++.

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	0 à 65 535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	$-3.4*10^{38}$ à $3.4*10^{38}$
double	Flottant double	8	$-1.7*10^{308}$ à $1.7*10^{308}$
long double	Flottant double long	10	$-3.4*10^{-4932}$ à $3.4*10^{4932}$
bool	Booléen	Même taille que le type *int*, parfois *1* sur quelques compilateurs	Prend deux valeurs: *true* et *false* mais une conversion implicite (valant 0 ou 1) est faite par le compilateur lorsque l'on affecte un

Tableau 1 : Types de variables en C++.

3.3. Déclaration des variables

En C++, toute variable doit être déclarée avant d'être utilisée et avant de l'utiliser, il faut indiquer le **type** de ce variable, **son nom** et sa **valeur**, c.-à-d. :

type Nom_de_la_variable < (valeur) > ;

Ou bien utiliser la même syntaxe que dans le langage C.

type Nom_de_la_variable < = valeur > ;

Exemple :

```
#include <iostream>
using namespace std;
int main()
{   string nom(" ali mohammed");
    int aget(20);
    float  moyGen(12.43);
    double pi(3.14159);
    bool estAdmis(true);
    char lettre('a');
   return 0;
}
```

3.4. Constantes

Les constantes ne peuvent pas être initialisées après leur déclaration et leur valeur ne peut pas être modifiée après initialisation. Elles doivent être déclarées avec le mot clé **const** et obligatoirement initialisées dès sa définition.

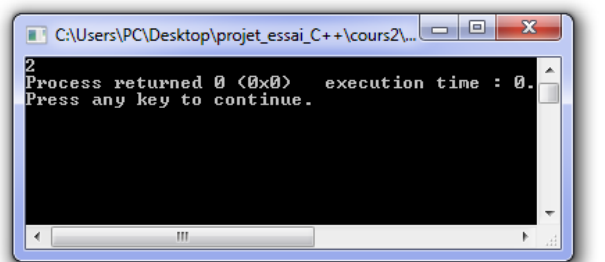
Syntaxe: const <type> <NomConstante> = <valeur>;

Exemple:

```
const char c ('ALI');          //exemple de constante caractère
const int i (2022);        //exemple de constante entiere
const double PI (3.14);      //exemple de constante rèel
const string motDePasse("pass_2022");      //exemple de constante chaine de caractères
```

Les constantes de type entier: **Enumérations**

```
#include <iostream>
 using namespace std;
 enum jour_semaine{sam,dim,lin,mar,mer,jeu,ver};
 int main()
{
    jour_semaine m=lin;
    cout<<m;

}
```



3.5. Assignation (Affectation)

Pour remplir les emplacements mémoires réservais à la déclaration on dispose soit d'une lecture directe au clavier, soit une affectation
Le signe (=) désigne l'assignation.

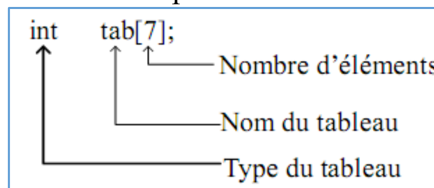
3.6. Initialisation des variables

Une valeur initiale peut être spécifiée dès la déclaration de la variable.
type variable = valeur ;
Exemple :

```
int i=1348;
float p=45.8765;
```

3.7. Tableaux

Un tableau est une suite séquentielle de cellules chacune d'elle contenant une donnée de même type. La taille d'un tableau est le nombre de ces cellules, cette taille doit être connue dès la déclaration du tableau. Un tableau peut être à une dimension ou à plusieurs dimensions. Il n'y a pas de limite (limitation physique de la mémoire).
Par exemple si on veut déclarer un tableau pouvant contenir 7 entiers, il faut :



Exemple

```
int      tab[7];
float   t[5] ;
double   vecteur [16] ;
```

Et pour un tableau à 2 dimensions :

```
int  Matrice [4][4] ;
 float   A[10][12] ;
```

Chaque élément d'un tableau est accédé par l'intermédiaire d'un indice (qui doit être un entier). L'indice 0 donne l'accès au premier élément. Ainsi pour mettre la valeur de 2 dans le premier et le dernier élément du tableau à 7 éléments.

```
int      tab[7];
 tab[0] =2 ;   tab[6] = 2 ;
```

Exemple Soit le tableau à 2 dimensions A[3][2] :

$$A\begin{pmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{pmatrix} \qquad \text{si } i=0 \quad A[i][i+1]=A[0][1]=2$$

```
#include <iostream>
 using namespace std;
  int main()

 //Access au element d'un tableau
   int tab2[3][2], i, j ;
   for (i=0; i<3; i++)
        for (j=0; j<2; j++)
        tab2[i][j] = i+j;
 for (i=0; i<3; i++)
        for (j=0; j<2; j++)
          cout << "tab2["<<i <<"][" << j<<"]="<< tab2[i][j] << endl;
 return 0 ;
   }
```

❖ **Initialisations des tableaux**

Lors de la déclaration d'un tableau, vous pouvez en profiter pour initialiser ses valeurs.

Exemple

int tableau [5] = {10, 20, 30, 40, 50} ;

Si vous précisez la taille du tableau, le compilateur produira un tableau contenant autant d'éléments que la liste.

int tab [] = {1, 2, 3, 4, 5} ; tableau à cinq éléments

• Il est interdit d'initialiser plus d'éléments que la taille du tableau

int tab [5] = {10, 20, 30, 40, 50, 60} ; erreur

• int tab [5] = {1, 2} ; déclaration d'un tableau à cinq éléments et initialisé uniquement les deux premiers tab[0] =1 et tab[1]=2.

• Les tableaux à plusieurs dimensions peuvent être initialisés, les valeurs sont affectées dans l'ordre des éléments.

int matrice[5][3] ={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15} ;

Pour plus de clarté, vous pouvez regrouper les différentes initialisations entre accolades.

int matrice[5][3] ={{1, 2, 3} ,{ 4, 5, 6}, { 7, 8, 9}, { 10, 11, 12}, { 13, 14, 15}} ;

3.8. Chaînes de Caractères

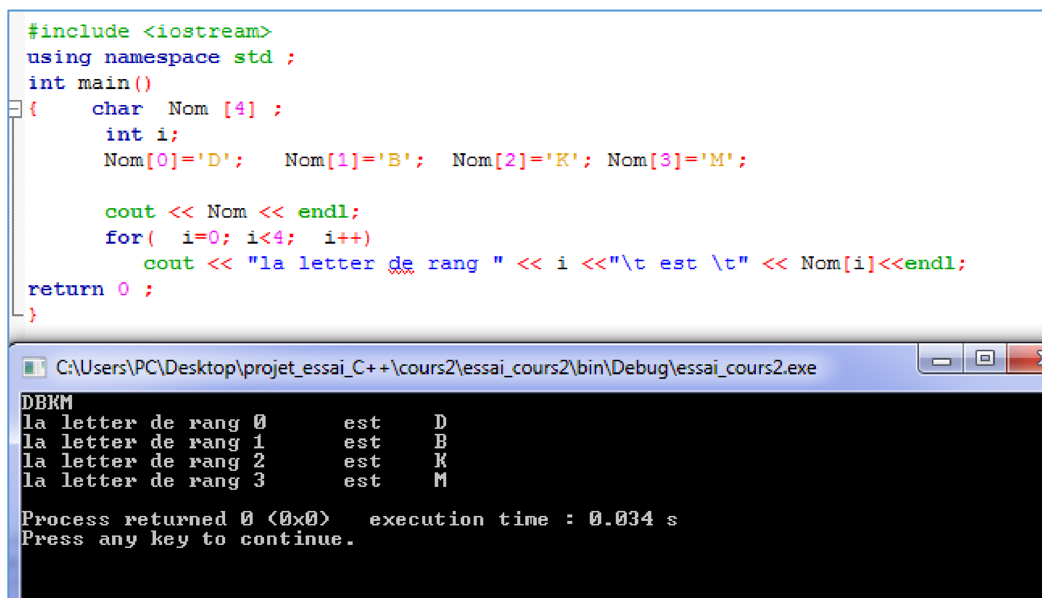
Une chaîne de caractère (string) est composée de caractères alphanumériques (texte). La déclaration d'une chaîne s'effectue par conséquent par spécification du nom set de la longueur de la chaîne. La longueur d'une chaîne correspond au nombre des caractères mentionné dans les crochets.

Exemple

char ch[10], ch1[100] ;

ch est la chaîne. ch[0], ch[1] des caractères.

$$char \quad c[5]; \quad \begin{cases} c[0]='H' & c[3]='L' \\ c[1]='E' & c[4]='O' \\ c[2]='L' & c="HELLO" \end{cases}$$



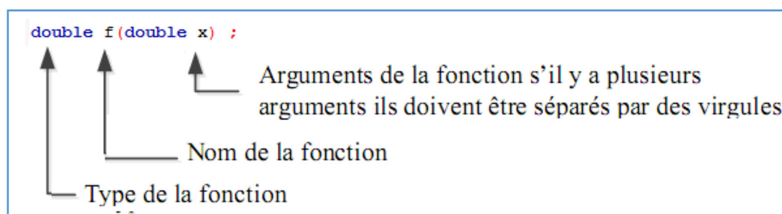
3.9. Les fonctions

La notion de prototype de fonction est un apport du langage C++ qui, comme la notion de constante, a migrée vers le langage C version ANSI. Elle permet de déclarer une fonction à l'intérieur d'une autre fonction (mais pas de la définir, un programme étant toujours une suite de définitions de fonctions).

Une fonction peut :

- Se trouver dans le même module de texte, être incluse automatiquement dans le texte du programme (par une directive #include) ou compilée séparément.
- Etre appelée à partir du programme principal, d'une autre fonction ou de la fonction elle-même (récursivité)
- Admettre ou non des arguments.
- Retourner ou non une valeur.
- Posséder ses propres variables. (variables locales)
- Une fonction à un seul point d'entrée (la première instruction de la fonction)
- Peut avoir plusieurs points de sorties par l'instruction return, ou automatiquement après la dernière instruction de la fonction.
- Ce ne fait aucune distinction entre procédure et fonction mais une procédure n'est rien d'autre qu'une fonction pour laquelle on ne se préoccupe pas de la valeur de retour sans type (void).
- C++ ne permet pas l'imbrication des fonctions, une fonction ne peut être déclarée à l'intérieur d'une autre fonction, mais une fonction peut appeler une autre fonction.
- Les arguments d'une fonction ne peuvent être passés que par valeur.

<u>Exemple :</u>



```
//déclaration d'un local
//===================================================

#include <iostream>
#include <math.h>

using namespace std;

int  main(void)
{
double f(double x) ;
float x, y ;
cout << "x = " ;
cin >> x ;
while (x != 1000)
{
y = f(x) ;
cout << "f(" << x << ") = " << y << "\n" ;
cout << "x = " ;
cin >> x ;
}
return 0;
}
double f(double x)
{
return((sin(x) + log(x))/(exp(x) + 2)) ;
}
```

3.8.1. Quelques règles sur les fonctions

- Les arguments figurent dans l'entête de la fonction se nomment arguments muets ou formels.
- Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment les arguments effectifs.
- L'instruction return peut mentionner n'importe quelle expression
- L'instruction return peut apparaître à plusieurs reprises dans une fonction. Elle peut apparaître à plusieurs reprises dans une fonction.

```
double   absom(double s)
{
  if  (s<0)   return s;
  else  return –s;
}
```

Quand une fonction ne renvoie pas de résultat, on le précise dans l'entête et dans la déclaration, à l'aide du mot clé void.

```
void affiche(int n) ;
```

3.10. Opérateurs et expressions

En combinant des variables, des opérateurs et de fonctions, on obtient des expressions.

❖ **Opérateurs arithmétiques**

+ : addition,
- : soustraction,
* : multiplication,
/ : division (19.0 / 5.0 vaut 3.8 et 19 / 5 vaut 3)
% : reste de la division entière de deux entiers (modulo)(19 % 5 vaut 4)
++ : incrémentation. Les expressions i+ + et ++i ont pour valeur :
– pour la première, d'ajouter ensuite 1 à la valeur de i (post-incrémentation),
– pour la seconde, d'ajouter d'abord 1 à la valeur de i (pré-incrémentation).
-- : décrémentation. i-- et --i fonctionnent comme i++ et ++i, mais retranchent 1 à la valeur de i au lieu d'ajouter 1.

<u>Exemple</u>

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
    int a, b=3, c, d=3; a=++b; // équivalent à b++; puis a=b; => a=b=4
 c=d++; // équivalent à c=d; puis d++; => c=3 et d=4
 cout<<"a="<<a<<endl;
 cout<<"c="<<c;
}
```

Dans les expressions arithmétiques, les règles de priorité sont les règles usuelles mathématiques. Par exemple, l'expression 5 + 3 * 2 a pour valeur 11 et non 16. En cas de doute, il faut mettre des parenthèses.

Il arrive très souvent de calculer la nouvelle valeur d'une variable en fonction de son ancienne valeur. C++ fournit pour cela un jeu d'opérateurs combinés, de la forme :
+=, -=, *=, /=, %=
Forme générale : variable opérateur= expression
L'expression est équivalente `a :
variable = variable opérateur expression
Par exemple, l'expression i += 3 équivaut `a i = i + 3.

❖ **Opérateurs logiques**

! : non,
&& : et,
|| : ou.

❖ **Opérateurs de comparaison**

== : égal,
!= : difféerent,
< : strictement inférieur,
> : strictement supérieur,
<= : inférieur ou égal,
>= : supérieur ou égal.

3. Les structures de contrôle

3.1 Structures de contrôle conditionnelles

3.1.1. L'instruction if

L'instruction if exécute de manière conditionnelle une instruction ou un bloc d'instructions. La condition est déterminée par la valeur d'une expression booléenne (par exemple :(i > 1)).
Si une seule instruction dépend de la condition, on écrit :

if (condition)
instruction; // exécute si condition est true
Ou
if (condition)
{
instruction; // ex¶ecut¶e si condition est true
}

Si plusieurs instructions dépendent de la condition, on écrit :

if (condition)
{
instruction1; // exécute si condition est true
instruction2;//
}

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


int main(int argc, char** argv) {
    int r,z;
    cin>>r;
if (r>=9)
 {
    cout<<"r est sup à 9"<<endl;
    z=2*r;
    cout<<"la valeur de z est "<<z<<endl;
}
return 0;
    }
```

3.1.2. L'instruction if ... else

L'instruction if ... else exécute de manière conditionnelle une instruction ou un bloc d'instructions. Si la condition n'est pas remplie une autre instruction ou bloc d'instructions est exécuté. La condition est une expression booléenne.

Si une seule instruction dépend de la condition, la syntaxe est la suivante :

if (condition)
instruction; // ou {instruction;} // exécute si condition est true
else
instruction; // ou {instruction;} // exécute si condition est false

Si plusieurs instructions dépendent de la condition, on écrit :

if (condition)
{
instructions; // exécute si condition est true
}
else
{
instructions; // exécute si condition est false
}

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

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


    int r,z;
    cin>>r;
if (r>9)
 {
    cout<<"r est sup a 9"<<endl;
}
else
{
cout<<"r est inf a 9"<<endl;
}

return 0;
    }
```

3.2. L'instruction switch

voici un exemple le fonctionnement de l'instruction switch :

```
int i;

switch (i)
{
    case 0:
        instructions;  // instructions à exécuter si i = 0 (noter qu'il n'y a pas de { })
        break;         // Attention, important, bien écrire cette instruction
    case 1:
        instructions;  // instructions à exécuter si i = 1
        break;
    case 2:
    case 3:
    case 4:
        instructions;  // instructions à exécuter si i = 2, 3 ou 4
        break;
    default:
        instructions;  // instructions à exécuter si i est différent de 0, 1, 2, 3 et 4
        break;         // le bloc default n'est pas obligatoire
}
```

Exemple

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

int main(int argc, char** argv) {
int i;
cin>>i;
switch (i)
{
case 0:
cout<<" cas initial"<<endl;
break;
case 1:
cout<<"1eme cas"<<endl;
break;
case 2:
cout<<"2eme cas"<<endl;
break;
case 3:
cout<<"3eme cas"<<endl;
break;
}
return 0;
}
```

3.3. Structures de contrôle itératives

3.3.1. L'instruction while

```
while (expression booléenne)
    instruction; // ou {instruction;} ou {instructions;}
```

Signifie tant que l'expression booléenne est vraie, exécuter l'instruction ou le bloc d'instructions.

```
#include <iostream>
using namespace std;
int main(int argc, char** argv) {
    int i=0;
    cout<<"exemple 1 while"<<endl;
    while (i<10){

    cout<<i<<endl;
    i++;
        }
return 0;
}
```

```
exemple 1 while
0
1
2
3
4
5
6
7
8
9
```

3.3.2. L'instruction do ... while

```
do
     instruction; // ou {instruction;} ou {instructions;}
while (expression booléenne);
```

Signifie exécuter l'instruction ou le bloc d'instructions, tant que l'expression booléenne est vraie. Par rapport au while, l'instruction ou le bloc d'instructions est exécuté au moins une foisavec le do.

```
#include <iostream>
using namespace std;
int main(int argc, char** argv) {

cout<<"exemple 2 while"<<endl;
        int j=11;

while (j<10){

    cout<<j<<endl;// rien à affiché
    j++;
        }


cout<<"exemple 2 do while"<<endl;

do {
cout<<j<<endl;
j++;
}
while (j<10);

return 0;
}
```

```
exemple 2 while
exemple 2 do while
11
```

3.3.3. L'instruction for

```
for (init_instruction; condition; reinit_instruction)
     instruction; // ou {instruction;} ou {instructions;}
```

L'instruction for fonctionne comme suit :
-L'instruction **init_instruction** est exécutée
-La condition (expression booléenne) **condition** est vérifie
-Si la condition est vraie, le bloc d'instructions est exécuté, sinon la boucle s'arrête
-L'instruction **reinit_instruction** est exécutée
-La condition **condition** est vérifie
-si la condition est vraie, . . .

Dans l'exemple suivant

```
for (i = 0; i < 10; i++)
     instruction; // ou {instruction;} ou {instructions;}
```

Le bloc d'instructions est exécuté 10 fois. La première fois, i vaut 0, puis 1, . . . jusque 9.
Exemple

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

int main() {
int i;
for (i=0;i<15;i++)
{
cout<<"iteration nemuro"<<i<<endl;

}

return 0;
}
```

```
iteration nemuro0
iteration nemuro1
iteration nemuro2
iteration nemuro3
iteration nemuro4
iteration nemuro5
iteration nemuro6
iteration nemuro7
iteration nemuro8
iteration nemuro9
iteration nemuro10
iteration nemuro11
iteration nemuro12
iteration nemuro13
iteration nemuro14
```

4. Ruptures de Séquence

4.1. Break

L'instruction break sert à "casser" ou interrompre une boucle (for, while et do … while), ou un switch. L'exécution reprend immédiatement après le bloc terminé.

Exemple :

```
//==========================================================================
// break
//==========================================================================

#include <iostream>
using namespace std;

int main()
{ int i;
  for (i = 0 ; i < 10 ; i++)
   { cout <<" i= " <<i<< endl;
     if (i==5)
       break;
   }
cout<<" Valeur de i est superieure de : " << i <<endl ; return 0;
}
```

```
C:\Users\PC\Desktop\projet_essai_C++\projet_devC++
i= 0
i= 1
i= 2
i= 3
i= 4
i= 5
Valeur de i est superieure de : 5
```

4.2. Continue

L'instruction continue sert à "continuer" une boucle (for, while et do … while) avec la prochaine itération.

Exemple

```
//==========================================================================
//==========================================================================
// continue
//==========================================================================
#include <iostream>
using namespace std;

int main()
{ int i;
  for (i = 0 ; i < 5 ; i++)
   { if (i==5)
       continue;
     cout <<" i= " <<i<< endl;
   }
cout<<" Valeur de i apres la fin de boucle : " << i <<endl ; return 0;
}
```

```
C:\Users\PC\Desktop\projet_essai_C++\projet_devC++\essai2.exe
i= 0
i= 1
i= 2
i= 3
i= 4
Valeur de i apres la fin de boucle : 5

--------------------------------
Process exited after 0.1743 seconds with return value 0
Appuyez sur une touche pour continuer...
```

5. Fonctions sans retour

C++ permet de créer des fonctions qui ne renvoient aucun résultat. Mais, quand on la déclare, il faut quand même indiquer un type. On utilise le type void. Cela veut tout dire : il n'y a vraiment rien qui soit renvoyé par la fonction.

Exemple

```
// fonction sans retour
//==========================================================================

#include <iostream>
using namespace std;

void presenteprogramme () ;

int main ()
{
   presenteprogramme();
   return 0;
}

void presenteprogramme ()
{
   cout << "ce programme …";
}
```

```
C:\Users\PC\Desktop\projet_essai...
ce programme à
--------------------------------
Process exited after 0.1739 seconds w
Appuyez sur une touche pour continuer
```

6. Les pointeurs

Le contenu d'une variable est stocké en mémoire RAM en un certain emplacement. Chaque emplacement de cette mémoire possède une adresse bien déterminée. Cette adresse ne représente rien d'autre que le numéro d'octet ou est stockée la variable. La mémoire RAM consiste en effet en une succession d'octets numérotés dont le nombre est déterminé par la taille

de la mémoire (par exemple 256 ‘méga’ de RAM signifie $256*1024^2 = 256*1\ 048576 \approx 256$ millions d'octets).
On obtient l'adresse d'une variable a par exemple en écrivant &a. Cette adresse peut être stockée dans une autre variable d'un type particulier : le type **pointeur**. Une variable de type pointeur reçoit comme contenu une adresse. Aucun autre type ne peut recevoir une adresse.
La déclaration d'une variable de type pointeur obéit à la syntaxe suivante.

- Si p veut recevoir par exemple l'adresse d'une variable de type double, on écrit : **double *p;**
- Si p devait recevoir l'adresse d'une variable de type int, on écrirait : **int *p;**
- Pour affecter au pointeur p l'adresse d'une variable a de type double, on écrira :

double a;
double *p;
p = &a

On dit que p pointe vers a. Apres l'instruction p = &a ;, l'expression *p représente le contenu de a, c'est-à-dire le contenu de la variable dont l'adresse est stockée dans p. Les expressions :
double a = 2.;
double c = a;
et
double a = 2.;
double c = *p;
produisent le même effet. Dans les 2 cas, le contenu de la variable a est stocké dans c qui vaut 2 après l'affectation. On peut aussi écrire l'instruction *p = 3. ;, après quoi la variable a contient la valeur 3.
On peut définir plusieurs variables pointeurs qui contiennent l'adresse de a :
double a = 2.;
double *p1;
double *p2;
p1 = &a;
p2 = p1; // OK parce que p1 et p2 pointent tous les 2 vers un double

7. Références

On peut coller plusieurs étiquettes à la même case mémoire (ou variable). On obtient alors un deuxième moyen d'y accéder. On parle parfois d'alias, mais le mot correct en C++ est référencé. Pour déclarer une référence sur une variable, on utilise une esperluette (&).
Exemple :

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


int main(){

int a(5);
   int &  b(a);

   int c=b ;
   cout<<" a= "<<a<< "\t "<<" b= "<<b<<" \t"<<"  c= "<<c<<endl;
   b=b+2;
   c=a ;
   cout<<" a= "<<a<< " \t "<<" b="<<b<<" \t "<<" c="<<c<<endl;

return 0;

}
```

Si la variable est définie dans un autre module il faut la déclarer avant de l’utiliser en utilisant le mot clé **extern**
Exemple

essai2.cpp

```
//============================
// Les références
//============================

#include <iostream>
#include"file1.cpp"
using namespace std;

 extern int H;

int main(){


   int limit;
   limit=H*2;// usage
   cout<<"H="<<H;

return 0;

}
```

file1.cpp

```
int H = 35;
```

8. Les conversions de type

Les opérateurs agissent sur des données de même type. Si on souhaite travailler avec des données de types différents, il y a lieu de convertir les types des données pour les rendre égaux. La conversion de type (ce que l'on appelle le typecasting) peut engendrer une perte de précision si on convertit une donnée vers un type moins précis.
Pour convertir une donnée d'un type donné vers un autre type, on écrit simplement le type souhaite entre parenthèses devant la donnée :

```
#include <iostream>
using namespace std;
/* run this program using the console pauser or add your own getch, system("pause") or input loop */

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

double a;
int i = 4, j;
a = (double)i * 2.4; /*i est converti en double, puis multiplie par le flottant 2.4 a vaut 9.6 au terme de cette opération*/
j = (int)a; // a est converti en entier (j ne contient plus que la partie entière de a)
            // j vaut 9 au terme de cette operation

cout<<a<<endl;
cout<<j;

    return 0;
}
```

Les tableaux ci-dessous résument les actions de conversion :

	Conversions sans perte de précision
`short`	→ `int, long`
`int`	→ `long`
`short`	→ `float`
`int, long`	→ `double`
`float`	→ `double`

		Conversions avec perte de précision
`int, long`	→ `short`	les nombres au-delà de 32767 sont perdus
`int, long`	→ `float`	perte de chiffres significatifs si le nombre est trop grand
`float`	→ `short`	seule la partie entière du `float` est conservée
`float, double`	→ `int, long`	seule la partie entière du `float, double` est conservée
`double`	→ `float`	perte de chiffres significatifs

9. Surcharge des fonctions

Plusieurs fonctions peuvent porter le même nom si leurs signatures diffèrent. La signature d'une fonction correspond aux caractéristiques de ses paramètres

- Leur nombre
- Le type respectif de chacun d'eux

Le compilateur choisira la fonction à utiliser selon les paramètres effectifs par rapport aux paramètres formels des fonctions candidates.

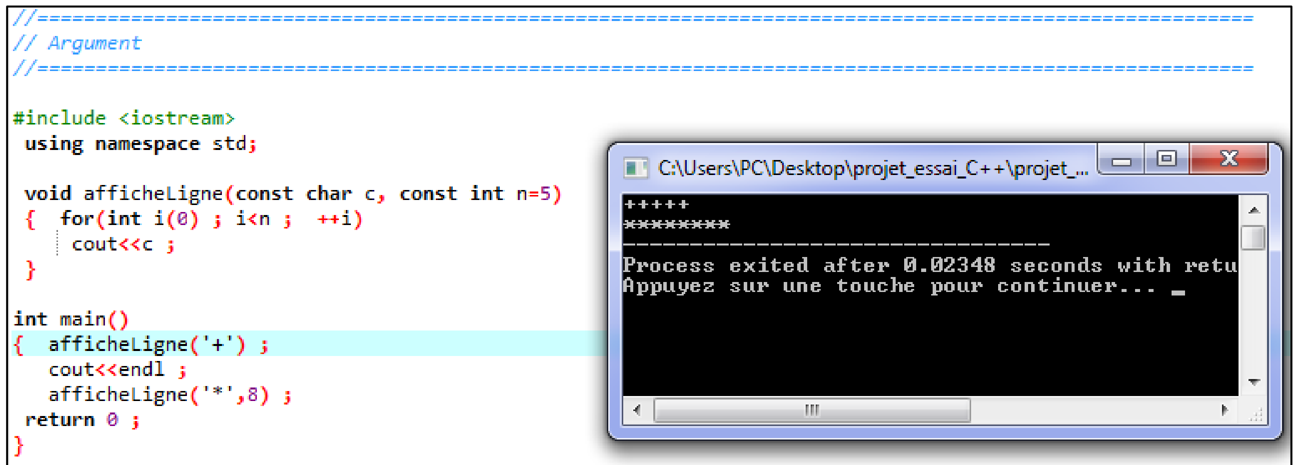
```
//=====================================================================
// surcharge des fonctions
//=====================================================================
#include <iostream>
 using namespace std;

 // définition de la fonction somme qui calcul la somme de deux réels

 double somme(double a, double b)
 {  double r;
    r = a + b;
    return r;
 }

 // définition de la fonction somme qui calcul la somme de deux entiers

double somme(int a, int b)
 {  double r;
    r = a + b;
    return r;
 }

 // définition de la fonction somme qui calcul la somme de trois réels

 double somme(double a, double b, double c)
 {  double r;
    r = a + b + c;
    return r;
 }


 int main()
{  double x, y,z, resultat;
   cout << "Tapez la valeur de x : "; cin >> x;
   cout << "Tapez la valeur de y : "; cin >> y;
   cout << "Tapez la valeur de z : "; cin >> z;

//appel de notre fonction somme (double,double)
  resultat = somme(x, y);
  cout <<  x << " + " << y << " = " << resultat << endl;

//appel de notre fonction somme (int,int)
  resultat = somme(static_cast<int>(x), static_cast<int> ( y));
  cout <<   static_cast<int>(x) << " + " << static_cast<int>( y) << " = " << resultat << endl;

//appel de notre fonction somme (double, double, double)
  resultat = somme(x, y,z);
 cout<<  x << " + " << y << " + " << z<< " = " << resultat << endl;

//appel de notre fonction somme (double, double, double)
  resultat = somme(static_cast<int>(x), static_cast<int>( y), static_cast<int>(z));
 cout<< static_cast<int> ( x) << " + " << static_cast<int>(y) << " + " << static_cast<int>( z)<< " = " <<
resultat << endl;
 return 0;
}
```

```
Tapez la valeur de x : 1.2
Tapez la valeur de y : 2.3
Tapez la valeur de z : 3.4
1.2 + 2.3 = 3.5
1 + 2 = 3
1.2 + 2.3 + 3.4 = 6.9
1 + 2 + 3 = 6

--------------------------------
Process exited after 8.77 seconds with ret
Appuyez sur une touche pour continuer...
```

10. Arguments par défaut

On peut, lors de la déclaration d'une fonction, donner des valeurs par défaut à certains paramètres des fonctions. Ainsi, lorsqu'on appelle une fonction, on ne sera pas obligé d'indiquer à chaque fois tous les paramètres.

Exemple



Remarques

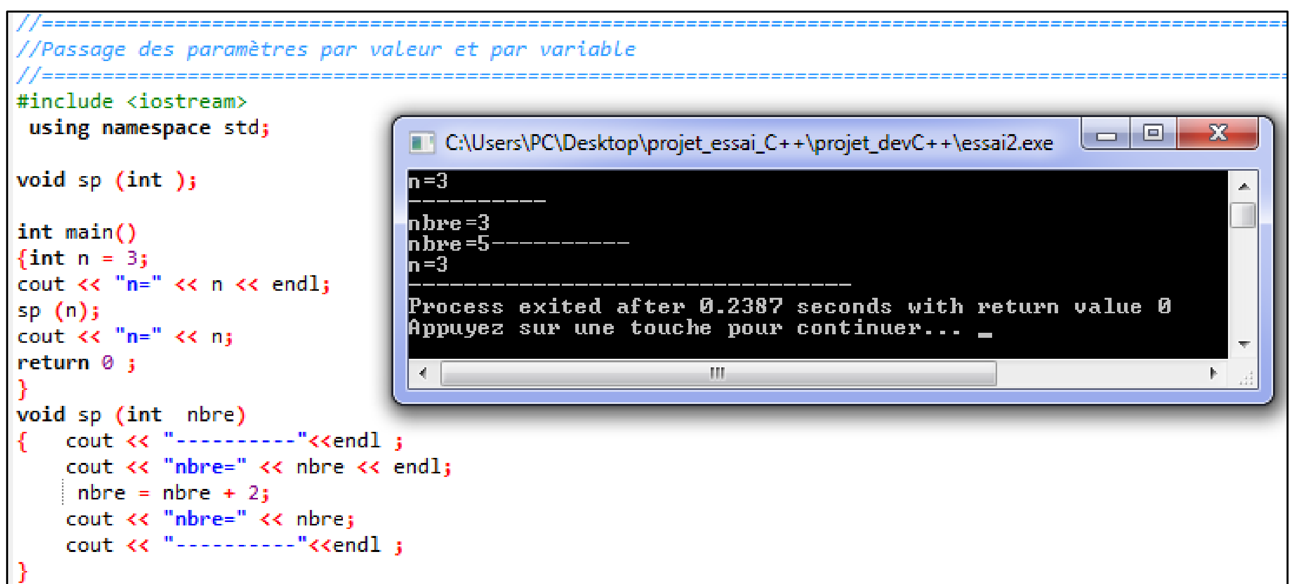
- Seul le prototype doit contenir les valeurs par défaut.
- Les valeurs par défaut doivent se trouver à la fin de la liste des paramètres.
- Vous pouvez rendre tous les paramètres de votre fonction facultatifs.

11. Passage des paramètres par valeur et par variable

11.1. Passage par valeur

La valeur de l'expression passée en paramètre est copiée dans une variable locale. C'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée.

Exemple



11.2. Passage par variable

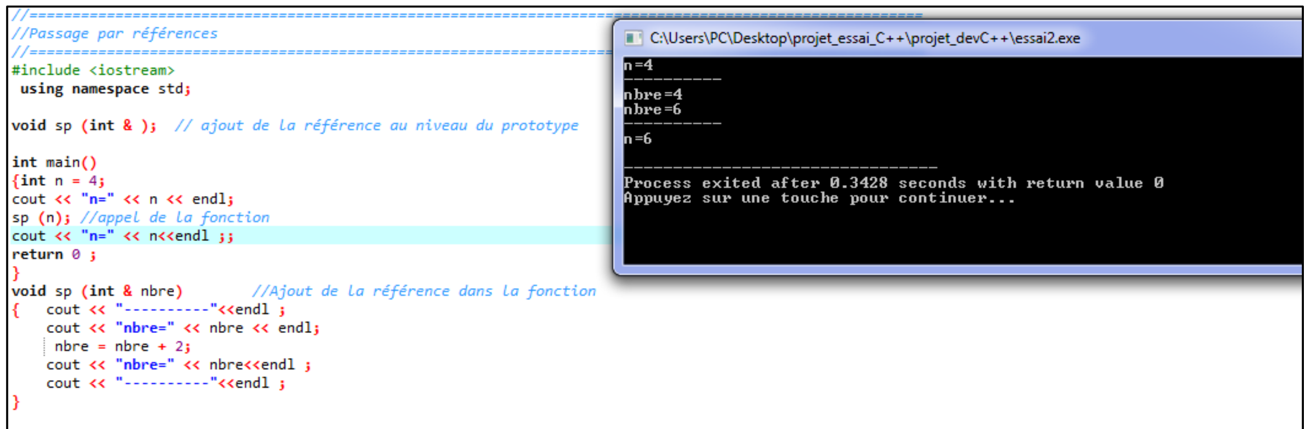
Consiste à passer l'adresse d'une variable en paramètre. Toute modification du paramètre dans la fonction affecte directement la variable passée en argument correspondant, puisque la fonction accède à l'emplacement mémoire de son argument. Il existe 2 possibilités pour transmettre des paramètres par variables :

- Les références
- Les pointeurs

a. Passage par références

Consiste à passer une référence de la variable en argument. Ainsi, aucune variable temporaire ne sera créée par la fonction et toutes les opérations (de la fonction) seront effectuées directement sur la variable. Le plus simple est d'utiliser le mécanisme de référence « & ».

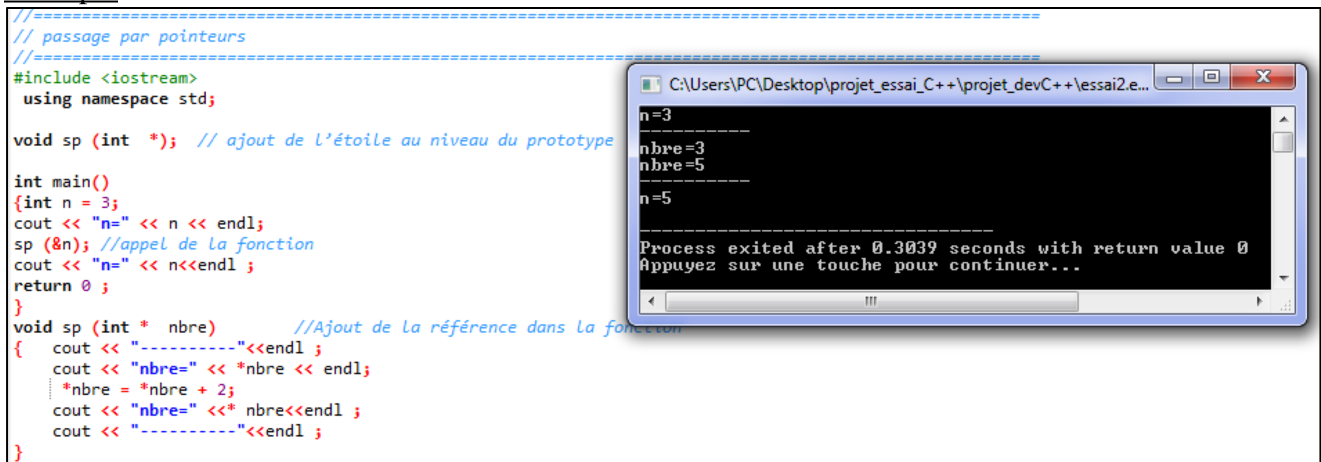
Exemple



b. Passage par adresses (pointeurs)

Consiste à passer l'adresse d'une variable en paramètre. Toute modification du paramètre dans la fonction affecte directement la variable passée en argument correspondant, puisque la fonction accède à l'emplacement mémoire de son argument. Le pointeur indique au compilateur que ce n'est pas la valeur qui est transmise, mais une adresse (un pointeur).

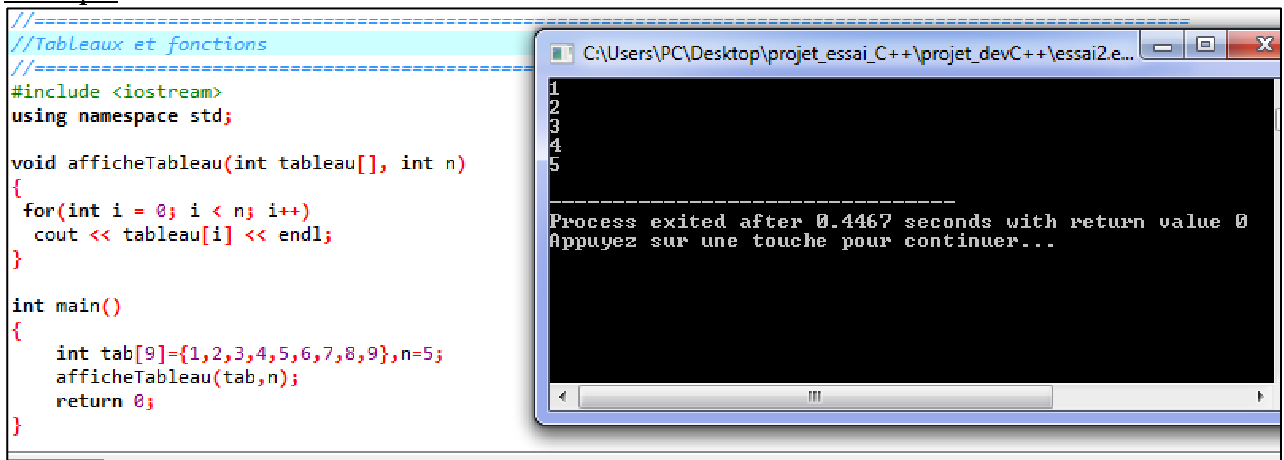
Exemple



12. Tableaux et fonctions

On peut passer un tableau comme argument d'une fonction. Voici donc une fonction qui reçoit un tableau en argument :

Exemple



13. Tableaux dynamiques

13.1. 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.

13.1.1. 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);**

Exemple

vector<int> tab(3);// pour un tableau de 3 entiers

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

vector<int> tab2(tab1);

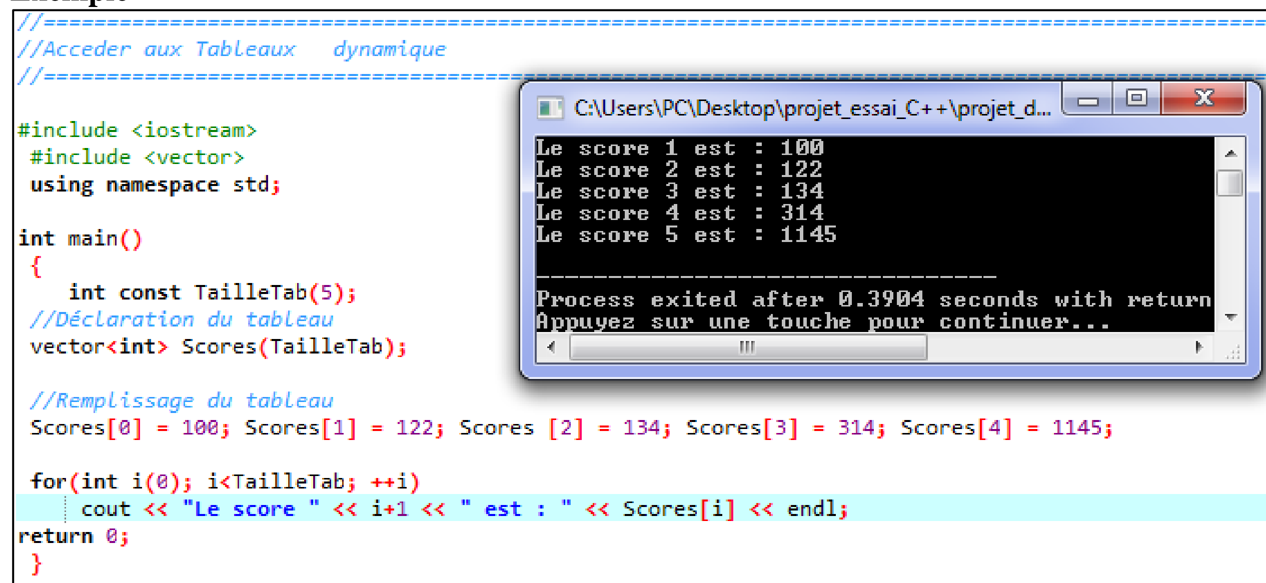
On peut même déclarer un tableau sans cases en ne mettant pas de parenthèses du tout:

vector<int> tab;

13.1.2. 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



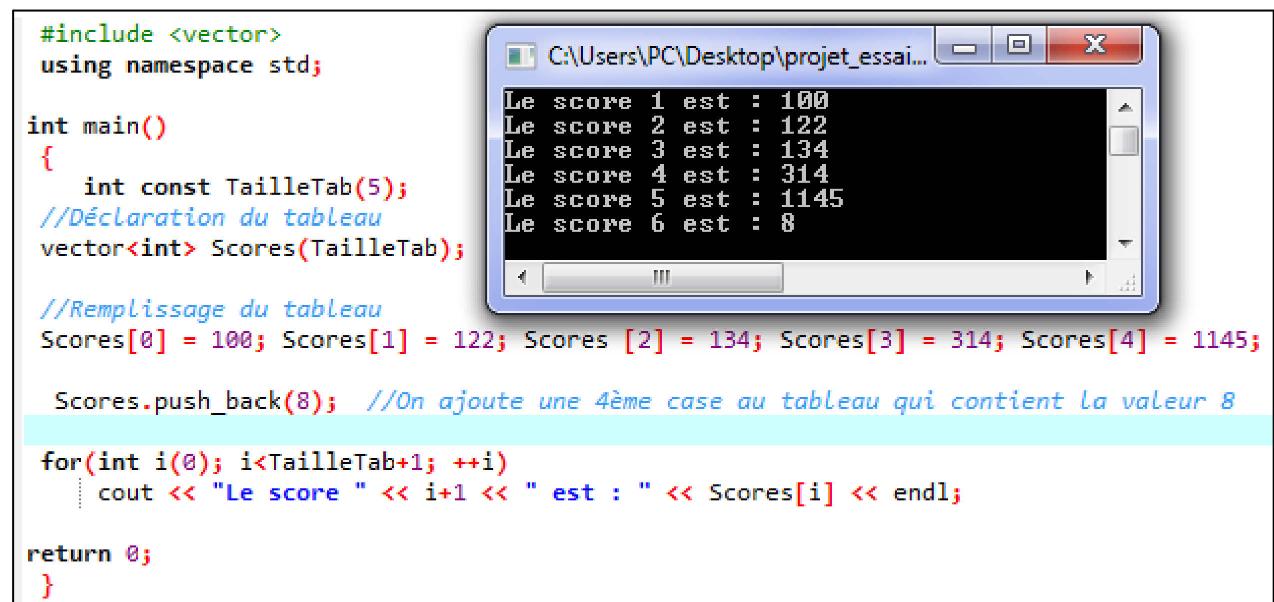
13.1.3. Modification de la taille

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

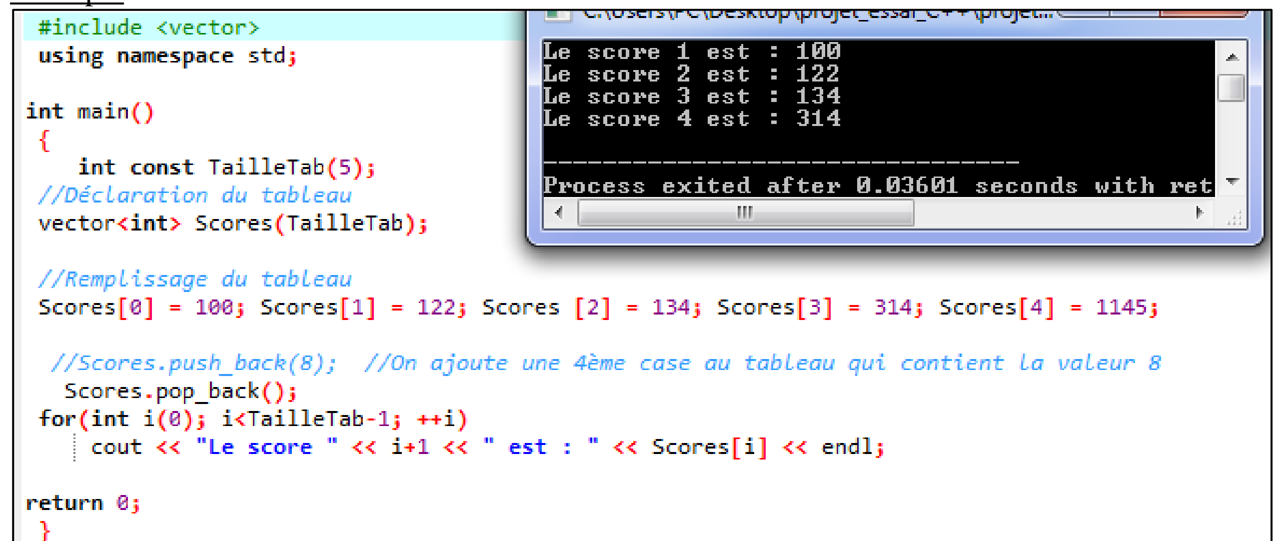


```
#include <vector>
using namespace std;

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

 //Remplissage du tableau
 Scores[0] = 100; Scores[1] = 122; Scores [2] = 134; Scores[3] = 314; Scores[4] = 1145;

  Scores.push_back(8);  //On ajoute une 4ème case au tableau qui contient la valeur 8

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

return 0;
 }
```


c. 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



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

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

 //Remplissage du tableau
 Scores[0] = 100; Scores[1] = 122; Scores [2] = 134; Scores[3] = 314; Scores[4] = 1145;

  //Scores.push_back(8);  //On ajoute une 4ème case au tableau qui contient la valeur 8
   Scores.pop_back();
 for(int i(0); i<TailleTab-1; ++i)
     cout << "Le score " << i+1 << " est : " << Scores[i] << endl;

return 0;
 }
```


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

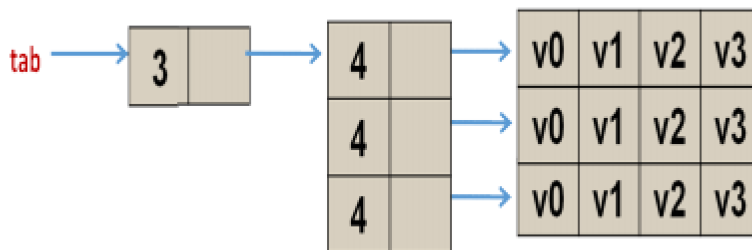
13.2. Tableaux multidimensionnels dynamiques

Nous pouvons créer des tableaux multidimensionnels de taille variable en utilisant des vecteurs
Pour un tableau à deux dimension contenant des valeurs entières, nous devons utiliser cette déclaration :

```
#include <iostream>
#include <vector>
 using namespace std;
 int main()
 {
    vector < vector<int> > tab;

    }
```

Ce programme montre un vecteur de vecteurs tab qui ne contient aucun élément.
Pour dimensionner ce vecteur de vecteurs, il faudra le faire en 2 fois

```
#include <vector>
 using namespace std;
 int main()
 {
    vector < vector<int> > tab;

    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


    }
```



Ce programme permet de créer un vecteur de 3 éléments, chaque élément désignant 1 vecteur de 4 éléments. Pour accéder aux valeurs du tableau, on utilise les parenthèses tab[i][j].

```
//=====================================================================================
//changement de la taille des tableaux multidimensionnels dynamiques
//=====================================================================================
#include <iostream>
#include <vector>
 using namespace std;
 int main()
 {
    vector<vector<int> > A;
A.push_back(vector<int>(5)); //ajouter une ligne de 5 cases à la matrice A
A.push_back(vector<int>(3,4)); // ajouter une ligne de 3 cases contenant chacune 4
A[0].push_back(8);// Ajouter une case contenant 8 à la première ligne de la matrice A


 }
```

13.3. Les strings (tableaux de caractères (lettres))

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.

Exemple : on représente la chaîne « Universite » de la manière suivante :

U	n	i	v	e	r	s	i	t	e	\0

13.3.1. Déclaration

Pour définir une chaîne de caractères, il suffit de définir un tableau de caractères. Le nombre maximum de caractères égal au nombre d'éléments du tableau moins un (réservé au caractère de fin de chaîne).

Exemple :

```
//==========================================================================================
// Les strings sont des tableaux de caractères, déclaration
//==========================================================================================

#include <iostream>
#include<string>
 using namespace std;
 int main ()
 {
char nom[20], prenom[20];    // 19 caractères utiles
char adresse[3][40];     // trois lignes de 39 caractères utiles

char texte[15] = {'U','n','i','v','e','r','s','i','t','e','\0'};// ou : char texte[15] = "Universite";

 }
```

Pour changer la longueur du texte, on utilise un vetor :

vector<char> texte;

13.3.2. Création d'objets « string »

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

```
//==========================================================================
// 12.2. Création d'objets " string "
//==========================================================================

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

13.3.3. Instanciation et initialisation de chaînes

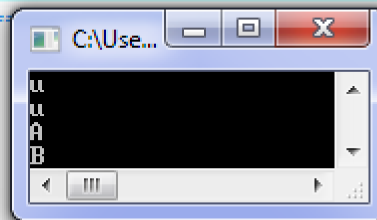
Il y a plusieurs possibilités pour initialiser l'objet au moment de la déclaration :

```
//==========================================================================================
//12.3. Instanciation et initialisation de chaînes
//==========================================================================================
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string ensemblechar("universite !");     //Création d'un objet 'ensemblechar' de type string et initialisation
    string s= "faculte";
    cout<<ensemblechar<<endl;
    cout<<s;

    return 0;
}
```

13.3.4. Accès à un caractère

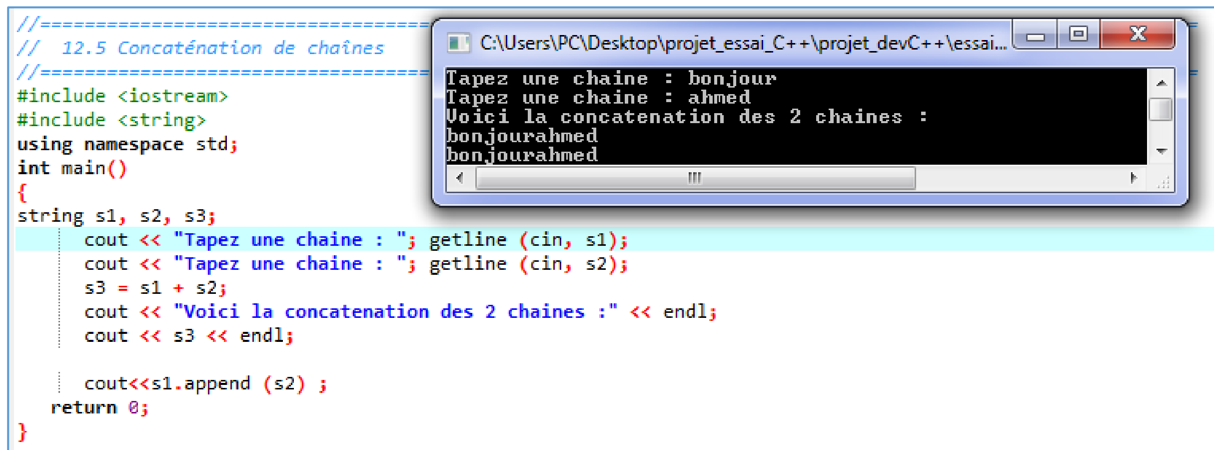
Pour accéder à un élément, on utilise les crochets.

```
//===========================================
//12.4. Accès à un caractère
//===========================================
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s= "faculte";
// lecture d'un caractère
cout << s[3]<<endl;        // 4eme caractère
cout << s.at(3)<<endl;   // 4eme caractère
// modification de caractère
s[2]     = 'A';
s.at(3)  = 'B';
cout<<s[2]<<endl;
cout<<s.at(3)<<endl;
}
```



14. Concaténation de chaînes

On peut assembler deux chaines de caractères:



```cpp
//==================================
//  12.5 Concaténation de chaînes
//==================================
#include <iostream>
#include <string>
using namespace std;
int main()
{
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;

     cout<<s1.append (s2) ;
   return 0;
}
```

```
C:\Users\PC\Desktop\projet_essai_C++\projet_devC++\essai...
Tapez une chaine : bonjour
Tapez une chaine : ahmed
Voici la concatenation des 2 chaines :
bonjourahmed
bonjourahmed
```

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

15. Quelques méthodes utiles du type « string »

Méthode size()

Pour déterminer la longueur de la chaîne stockée dans l'objet de type string, on utilise méthode size().

Méthode « erase() »

Pour supprimer le contenu de la chaine, on utilise la méthode erase()

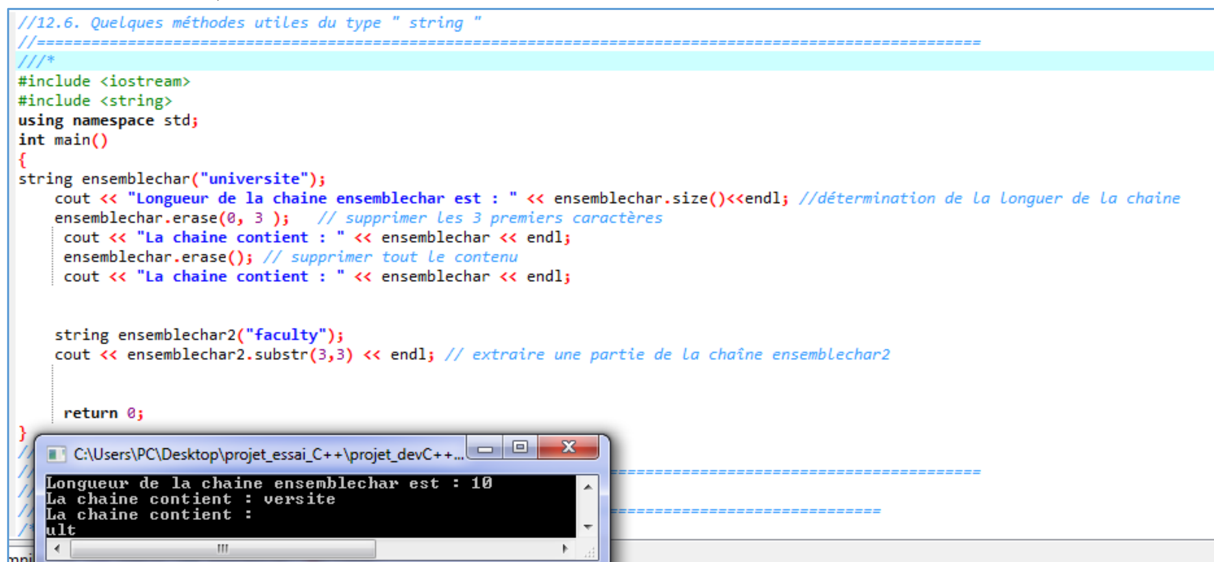
Méthode « substr() »

Cette méthode permet d'extraire une partie de la chaîne.

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

Index : permet d'indiquer à partir de quel caractère vous devez couper.

Num : permet d'indiquer le nombre de caractères à prendre (Par défaut, la valeur de npos prend tous les caractères restants.)



```cpp
//12.6. Quelques méthodes utiles du type " string "
//==========================================================
///*
#include <iostream>
#include <string>
using namespace std;
int main()
{
string ensemblechar("universite");
    cout << "Longueur de la chaine ensemblechar est : " << ensemblechar.size()<<endl; //détermination de la longuer de la chaine
    ensemblechar.erase(0, 3 );   // supprimer les 3 premiers caractères
     cout << "La chaine contient : " << ensemblechar << endl;
     ensemblechar.erase(); // supprimer tout le contenu
     cout << "La chaine contient : " << ensemblechar << endl;


    string ensemblechar2("faculty");
    cout << ensemblechar2.substr(3,3) << endl; // extraire une partie de la chaîne ensemblechar2


     return 0;
}
```

```
C:\Users\PC\Desktop\projet_essai_C++\projet_devC++...
Longueur de la chaine ensemblechar est : 10
La chaine contient : versite
La chaine contient :
ult
```

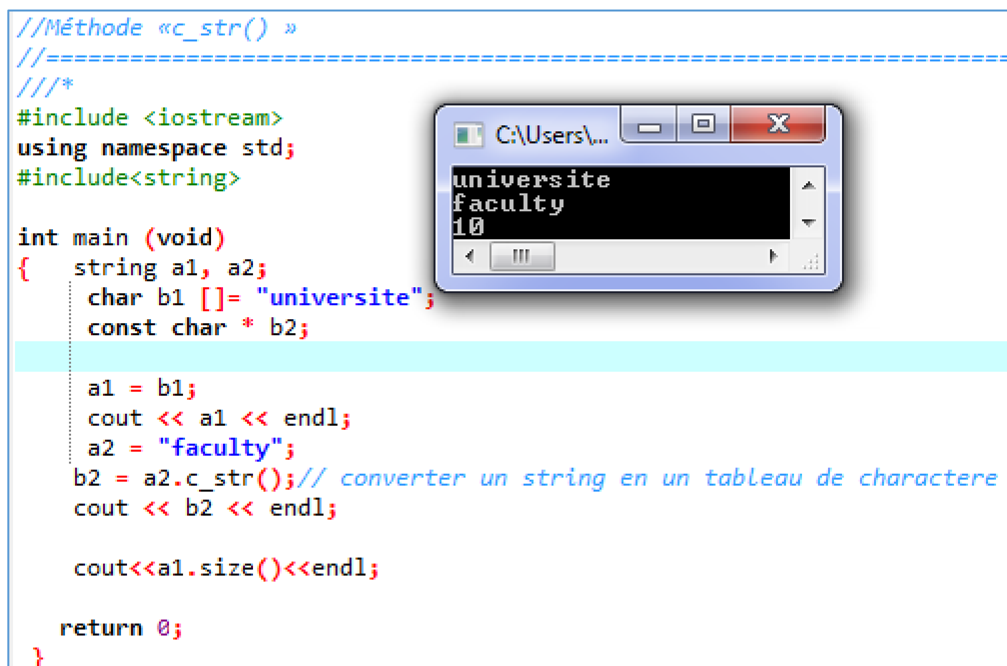
Méthode «c_str() »

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

-Le pointeur b2 est un pointeur vers un tableau non modifiable de char.

-Les variables a1 et a2 sont des string. On peut affecter directement a1=b1 : le tableau de char sera transformé en string.

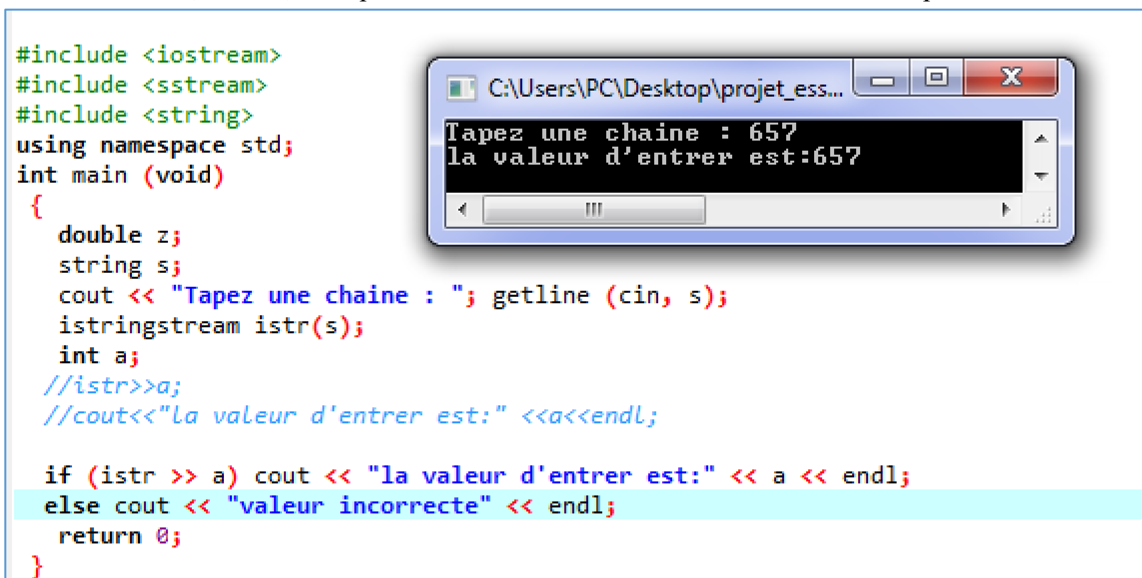
- On peut transformer aisément un string en tableau de char et inversement en écrivant b2=a2.c_str().



```cpp
//Méthode «c_str() »
//=====================================================================
///*
#include <iostream>
using namespace std;
#include<string>

int main (void)
{   string a1, a2;
     char b1 []= "universite";
     const char * b2;

     a1 = b1;
     cout << a1 << endl;
     a2 = "faculty";
    b2 = a2.c_str();// converter un string en un tableau de charactere
    cout << b2 << endl;

    cout<<a1.size()<<endl;

   return 0;
 }
```

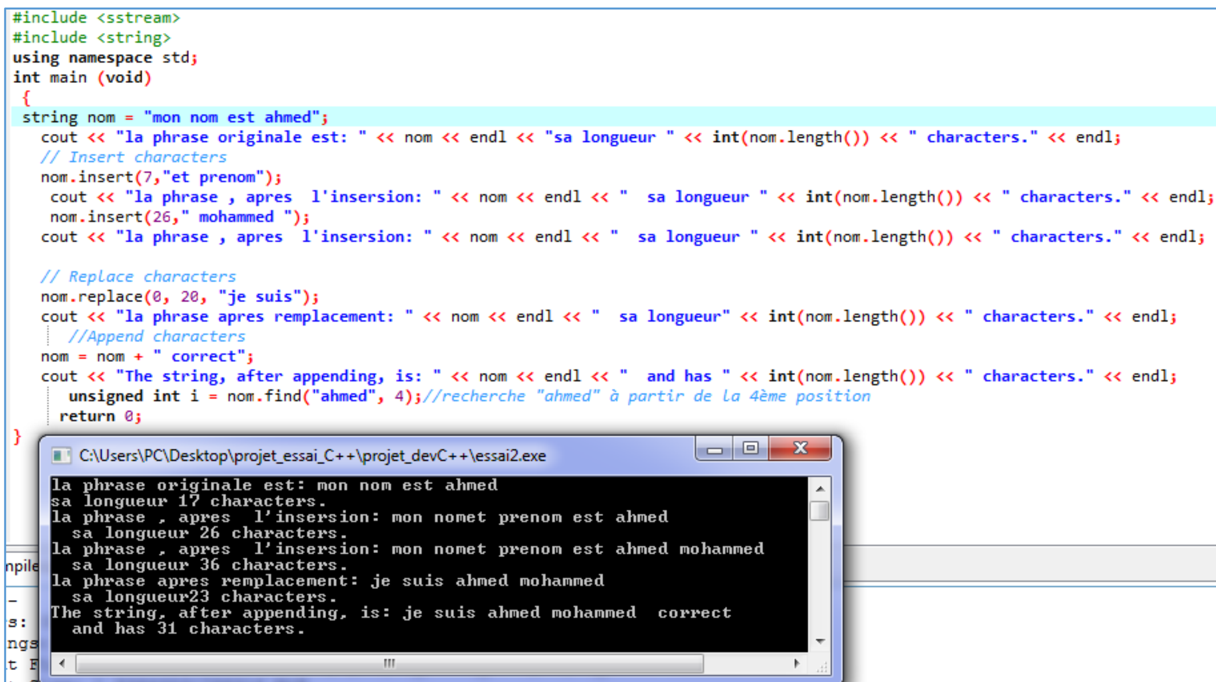

Méthode «istr() »

Pour transformer une chaîne en double ou en int, on utilise istringstream pour transformer la chaine en flot de sortie caractères, ensuite, nous pourrons lire ce flot de caractères en utilisant les opérateurs usuels >>.



```cpp
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main (void)
 {
   double z;
   string s;
   cout << "Tapez une chaine : "; getline (cin, s);
   istringstream istr(s);
   int a;
  //istr>>a;
  //cout<<"la valeur d'entrer est:" <<a<<endl;

  if (istr >> a) cout << "la valeur d'entrer est:" << a << endl;
  else cout << "valeur incorrecte" << endl;
   return 0;
 }
```


Les opérations : insert, replace, find



16. Allocation dynamique

Allocation statique : La réservation mémoire est déterminée à la compilation. L'espace alloué statiquement est déjà réservé dans le fichier exécutable du programme lorsque le système d'exploitation charge le programme en mémoire pour l'exécuter. L'allocation statique permet d'éviter les coûts de l'allocation dynamique à l'exécution.

Allocation dynamique: La mémoire est réservée pendant l'exécution du programme. L'espace alloué dynamiquement ne se trouve pas dans le fichier exécutable du programme lorsque le système d'exploitation charge le programme en mémoire pour l'exécute.

Exemple : les tableaux de taille variable (vector), les chaînes de caractères de type string.

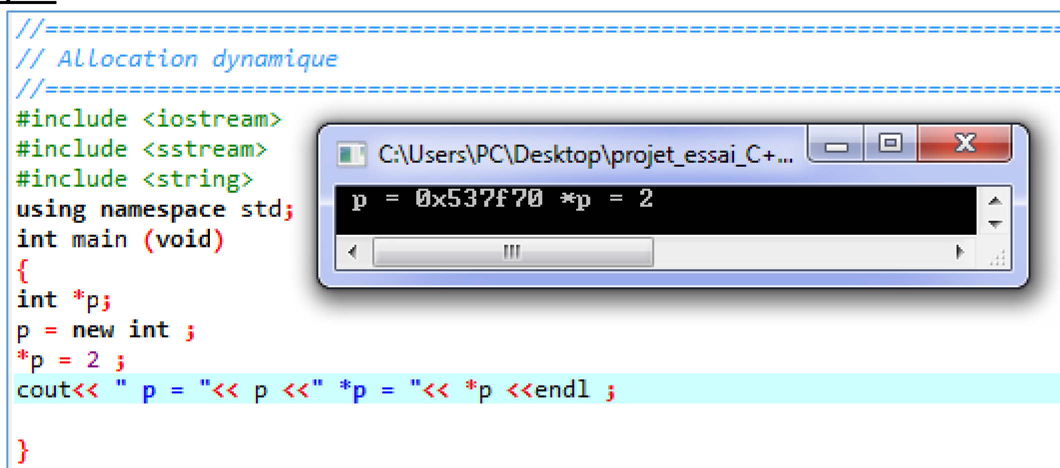
Pour les pointeurs, l'allocation dynamique permet également de réserver de la mémoire indépendamment de toute variable : on pointe directement sur une zone mémoire plutôt que sur une variable existante.

16.1. Allocation d'une case mémoire

Pour allouer et libérer dynamiquement de la mémoire, nous utilisons les opérateurs suivants : **new** et **delete.**

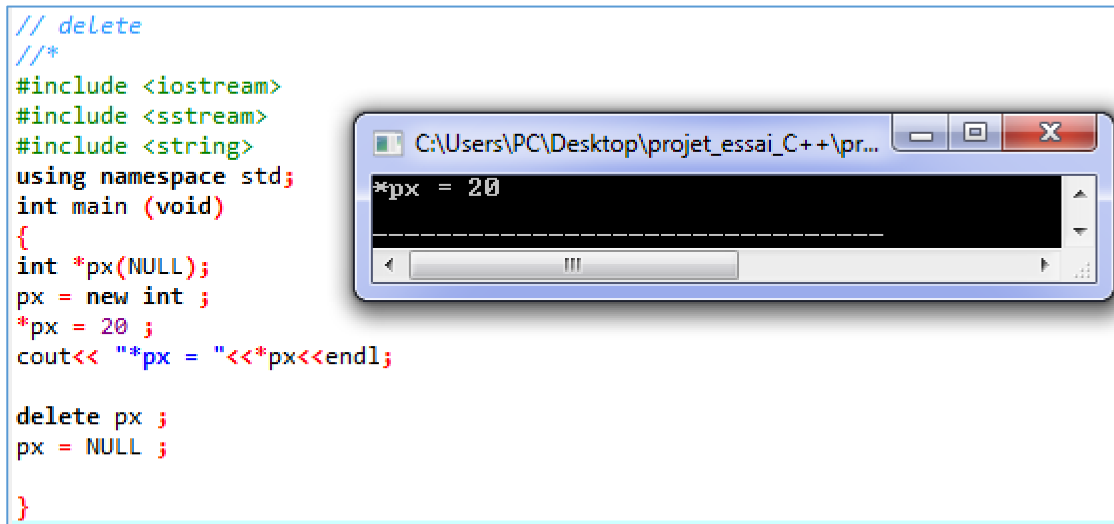
Syntaxe: **pointeur = new type** (Réserve une zone mémoire de type **type** et affecte l'adresse dans la variable pointeur.)

Exemple :

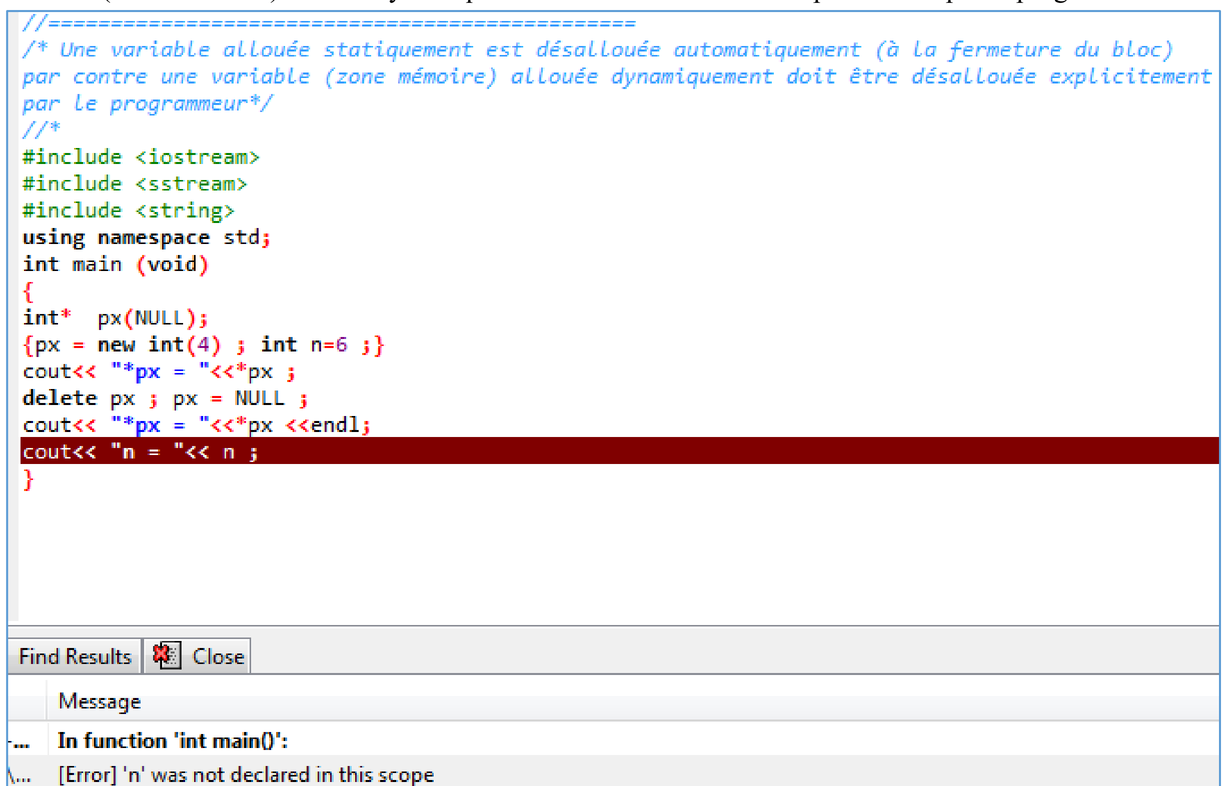


16.2. Libérer la mémoire allouée

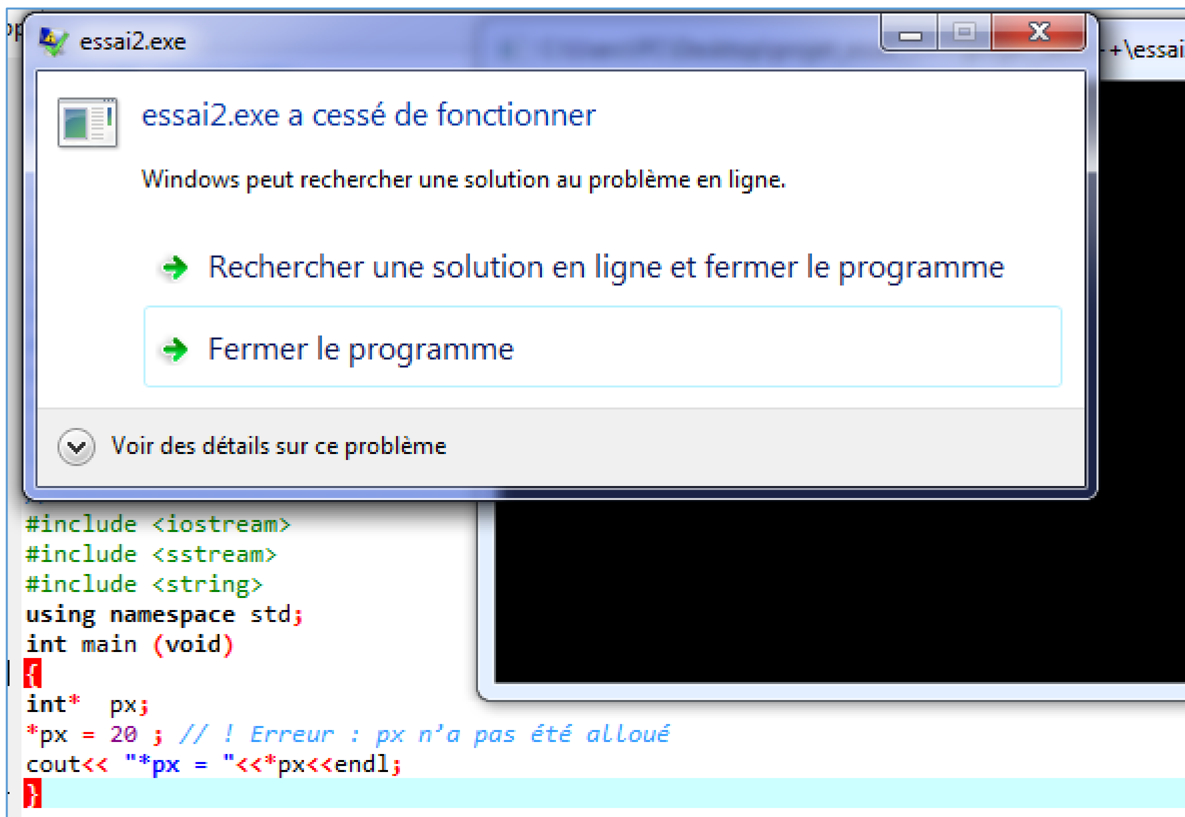
Pour libérer la zone mémoire allouée au pointeur, nous utilisons la syntaxe suivante: **delete pointeur**



Une variable allouée statiquement est désallouée automatiquement (à la fermeture du bloc) par contre une variable (zone mémoire) allouée dynamiquement doit être désallouée explicitement par le programmeur.



Si on essaye d'utiliser la valeur pointée par un pointeur pour lequel aucune mémoire n'a été réservée, une erreur de type **Segmentation fault** se produira à l'exécution. Le programme peut être compilé mais l'exécution a été arrêtée.



Si vous ne connaissez pas la mémoire pointée au moment de l'initialisation, utilisez NULL pour initialiser les pointeurs

```
//=================================================
//initialisation des pointeurs
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main (void)
{
int*  px(NULL);
if(px != NULL)
{   *px = 20 ;
    cout<< "*px = "<<*px<<endl;
}
}
```