# Programmation orientée Objet et C++ pour Eai

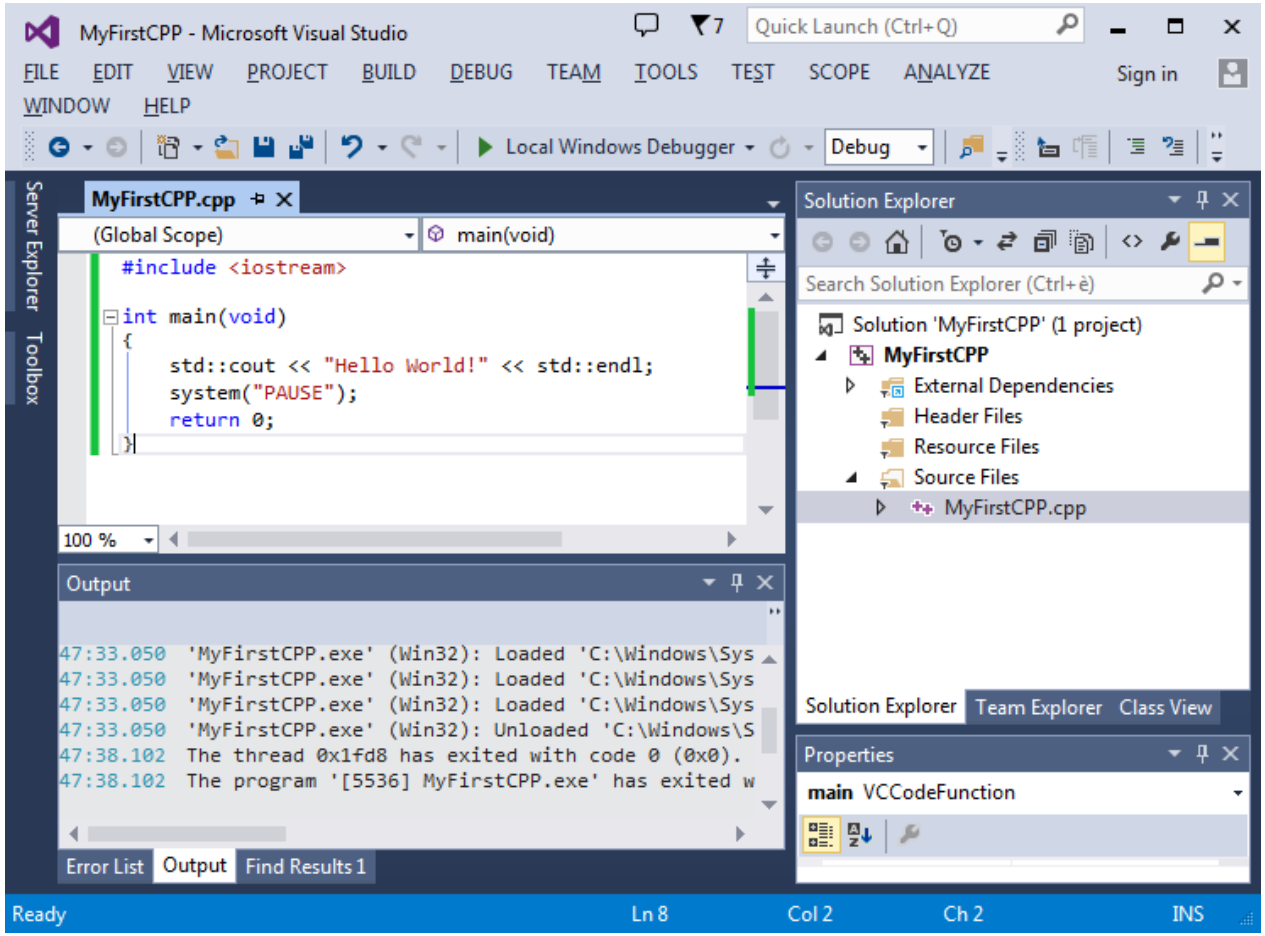

heig-vd

# Créer un programme C++ avec Visual Studio

- Donner l'extension .cpp au fichier source !

File-> New->
Project->Visual C++->
Win32 Console Applicat..

Choisir "empty project"



heig-vd

# Explications de "Déclaration" et "Définition"

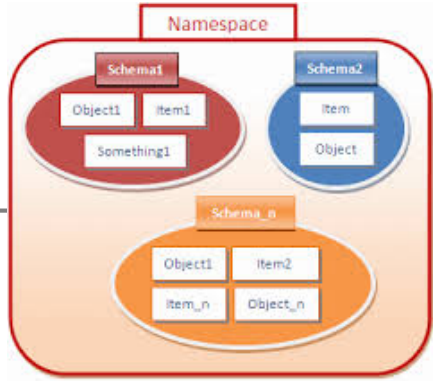
- Une déclaration introduit un nom - un identifiant - pour le compilateur. Elle indique au compilateur: "Cette fonction ou cette variable existe quelque part, et voici à quoi elle devrait ressembler."
- Les déclarations sont souvent dans les fichiers *.h

```
typedef struct {
   long largeur, hauteur;
   RGB * pixels;
} IMAGE;

int func1(int taille, int largeur);
```

- Une définition, d'un autre côté, dit : "Faire cette variable ici" ou "Faire cette fonction ici".
- Les définitions sont toujours dans les fichiers *.cpp

```
IMAGE cervin;

int func1(int taille, int largeur)
{
   …
}
```

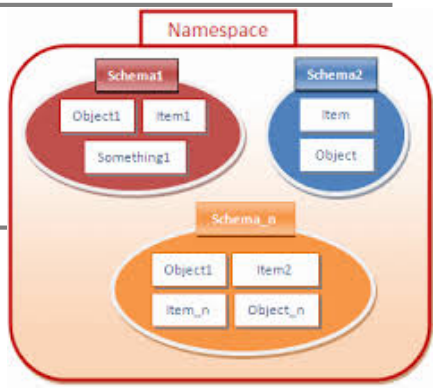
# Les espaces de nommage - utilisation



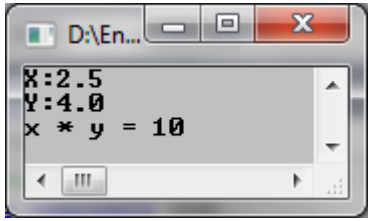

- Intérêt des espaces de nommage (namespace)
  - Grands projets avec de nombreuses bibliothèques
    - On pourrait avoir plusieurs bibliothèques contenant des fonctions différentes avec des noms identiques
    - Impossibilité de construire un tel programme en C
    - Permet d'éviter des "collisions" de noms
  - Espace de nom
    - Préfixe tous les identificateurs (fonctions, types, etc) de l'espace avec un nom
    - Exemple dans l'espace "std": std::cout
  - Simplification d'écriture
    - Il est possible d'importer un espace de nommage
    - Permet d'utiliser les identificateurs de cet espace sans préfixe
    - Syntaxe

```
using namespace std;
```

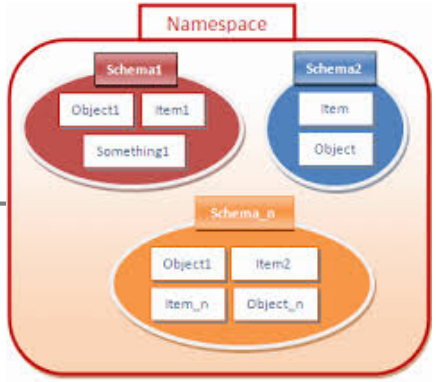
# namespace – utilisation - exemple




```
#include <iostream>
#include <conio.h>  // for _getch()
#include <cstdlib>  // for EXIT_SUCCESS

// Importation de l'espace de nom :
using namespace std;

int main(void)
{
    double x, y;

    // Le préfixe n'est plus requis :
    cout << "X:";
    cin >> x;
    cout << "Y:";
    // Il est toujours possible d'utiliser le préfixe :
    std::cin >> y;
    cout << "x * y = " << x * y << endl;
    // Les fonctions de la bibliothèque C sont toujours utilisables
    _getch();
    return EXIT_SUCCESS;
}
```



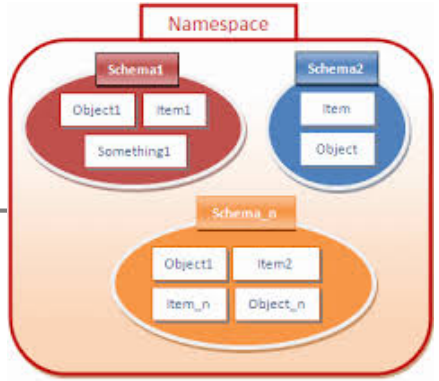
# namespace - définition




- Il est possible de définir un espace de nommage
- Les identificateurs de cet espace sont alors préfixés
- Les déclarations de namespace se mettent dans les fichiers header (.h ou .hpp) ET dans les fichiers source (.cpp)
- Syntaxe

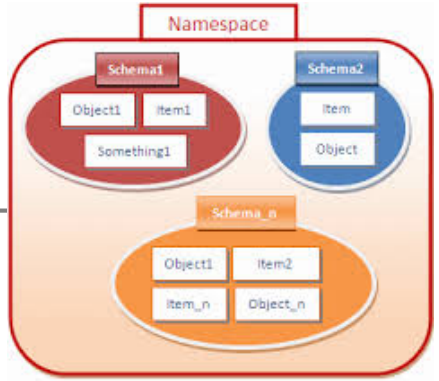
```
namespace nom
{
    // Placer ici les déclarations faisant partie de
    // l'espace de nom
}
```

# namespace – définition - exemple




```
namespace heig_vd
{
    void afficher_adresse()
    {
        cout << "HEIG-VD" << endl;
        cout << "Route de Cheseaux 1" << endl <<
                "1401 Yverdon les bains" << endl;
    }

    void afficher_coordonnees_completes()
    {
        // Préfixe non requis, car dans le même espace de nom :
        afficher_adresse();
        cout << "Tel : 024 55 76 330 \n";
    }
}

int main(void)
{
    heig_vd::afficher_coordonnees_completes();
    return EXIT_SUCCESS;
}
```

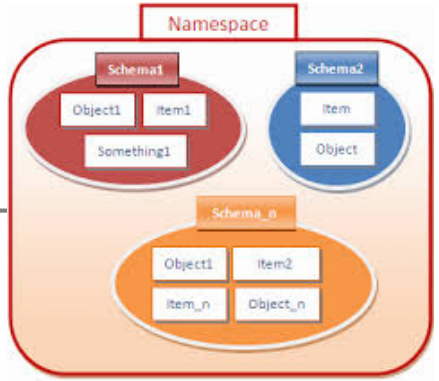
# Librairie standard du C++




- La librairie C++ inclut les mêmes définitions et fonctions que le C avec les différences suivantes:
  - Chaque fichier header a le même nom, mais préfixé avec "c" et sans le .h . Exemple: <stdlib.h> devient <cstdlib>
  - Chaque élément de la librairie doit être défini avec le namespace "std"
- Nouveau fichiers header
  - <string> Pour opérations sur les strings (pas string.h)
  - <iostream> Pour opération sur les flux

heig-vd

# Librairie standard du C++

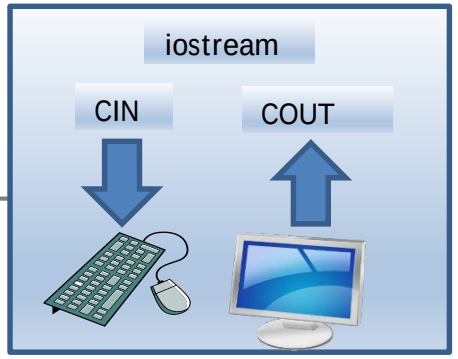



- Pour compatibilité, on peut encore utiliser les mêmes fichiers header que le C, mais ce n'est PAS recommandé

| | | |
|---|---|---|
| <cctype> | (ctype.h) | Character handling functions (header) |
| <cerrno> | (errno.h) | C Errors (header) |
| <cfloat> | (float.h) | Characteristics of floating-point types (header) |
| <climits> | (limits.h) | Sizes of integral types (header) |
| <cmath> | (math.h) | C numerics library (header) |
| <cstdbool> | (stdbool.h) | Boolean type (header) |
| <cstddef> | (stddef.h) | C Standard definitions (header) |
| <cstdio> | (stdio.h) | C library to perform Input/Output operations (header) |
| <cstdlib> | (stdlib.h) | C Standard General Utilities Library (header) |
| <cstring> | (string.h) | C Strings (header) |
| <ctime> | (time.h) | C Time Library (header) |

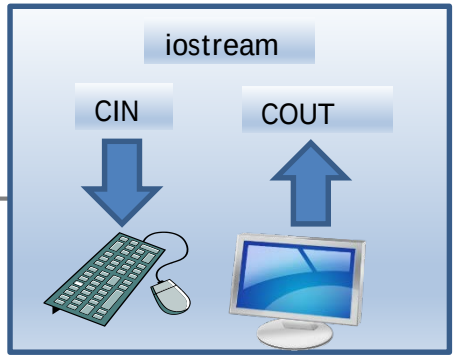
```
#include <iostream>
#include <cstdlib>
#include <string>
#include <limits>
```

# E/S C++ - Introduction




- Entrées/Sorties (E/S) C++
- <iostream> est la librairie C++ pour les entrées/sorties
- <iostream.h> est **l'ancienne** librairie C++ pour les E/S
- Lecture des données: cin
- Affichage des données: cout
- Pas de spécification de type (pour les types prédéfinis)
- Les E/S peuvent être redéfinies pour les types définis par l'utilisateur
- Un programme C++ peut aussi utiliser les E/S du C avec la librairie <cstdio> (exemples: printf, scanf, fopen, etc)

# E/S C++ - cout

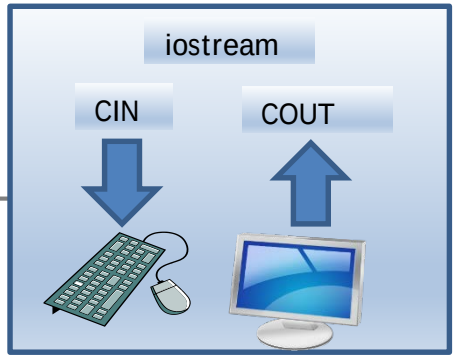



- Prototype de cette fonction

```
cout << arg1 << arg2 .. << argn;
```

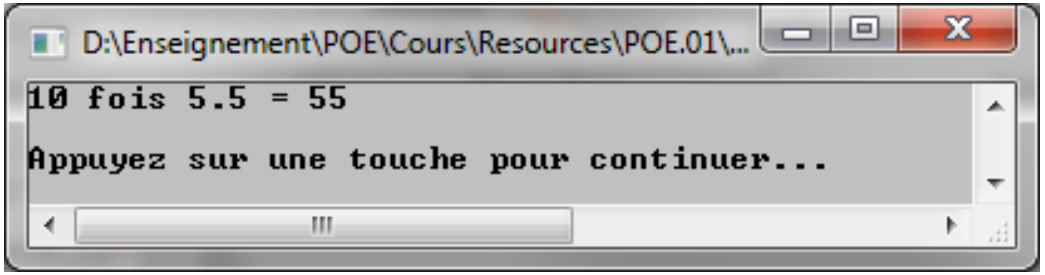
- Signification
  - Le flot standard de sortie est l'objet prédéfini: cout
  - La liste d'arguments peut être constituée d'une ou plusieurs expressions (types prédéfinis: int, double, string, etc)
  - L'opérateur (<<) appliqué à un "stream" est appelé: "opérateur d'insertion"
  - L'opérateur d'insertion (<<) peut être chainé

# E/S C++ - cout

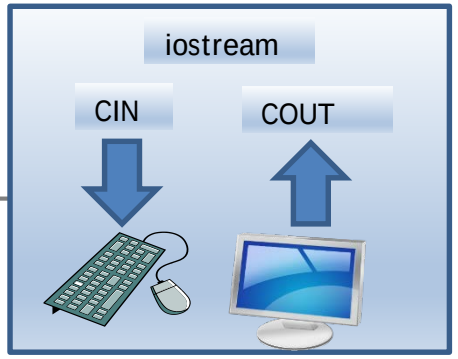



- Exemples de cout

```
double d(5.5);
int i(10);
cout << i << " fois " << d << " = " << i*d << "\n";
```



# E/S C++ - cin

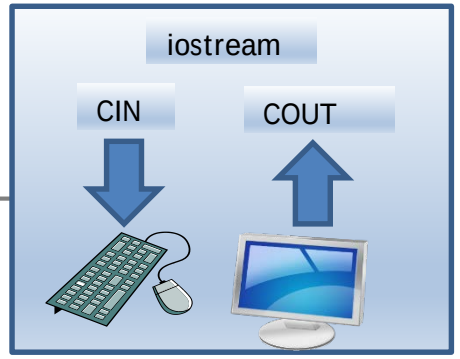



- Prototype de cette fonction

```
cin >> arg1 >> arg2 .. >> argn;
```

- Signification
  - Le flot standard d'entrée est l'objet prédéfini: cin
  - La liste d'arguments peut être constituée d'une ou plusieurs expressions (types prédéfinis: int, double, string, etc)
  - L'opérateur (>>) appliqué à un "stream" est appelé: "opérateur d'extraction"
  - L'opérateur d'extraction (>>) peut être chainé
  - cin laisse le caractère "newline" dans le flux (stream)

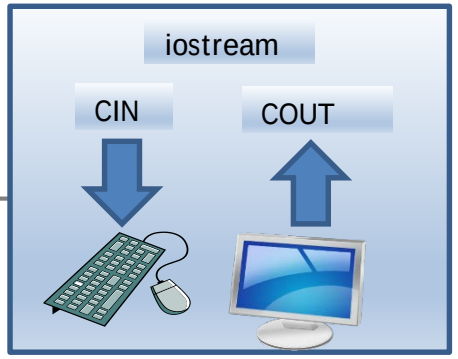
# E/S C++ - cin



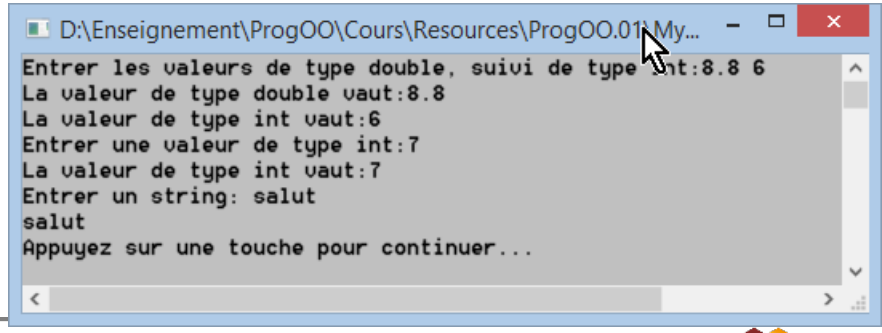

- Exemples de cin

```
double d = 5.5; int i = 10; string myString;
cin >> d >> i;
cin >> i;
cin >> myString; // lit UN string (s'arrête avec l'espace)
cin.ignore();  // flush 1 caractère de stdin buffer
getline(cin, myString); // met toute une ligne dans myString
```

# E/S C++ - cin




```
int main(void)
{
    double d = 5.5; int i = 10;
    string myString;
    system("COLOR 70");
    cout << "Entrer les valeurs de type double, suivi de type int:";
    cin >> d >> i;
    cout << "La valeur de type double vaut:" << d << endl;
    cout << "La valeur de type int vaut:" << i << endl;
    cout << "Entrer une valeur de type int:";
    cin >> i;
    cout << "La valeur de type int vaut:" << i << endl;
    cout << "Entrer un string: ";
    cin.ignore();  // flush 1 caractère de stdin buffer
    getline(cin, myString);
    cout << myString << endl;

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



# E/S C++ - exemple

```
#include <iostream>

int main(void)
{
    double x, y;

    std::cout << "X:";
    std::cin >> x;
    std::cout << "Y:";
    std::cin >> y;
    std::cout << "x * y = " << x * y << std::endl;
    return EXIT_SUCCESS;
}
```

Bibliothèque d'entrées sorties du C++

Flux de sortie

Opérateur surchargé en C++. Ecriture vers un flux.

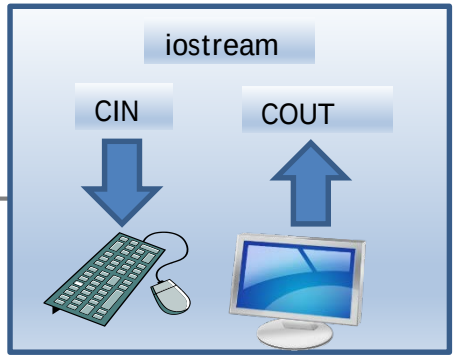
Flux d'entrée.

Opérateur surchargé en C++. Lecture depuis un flux.

Opérateur pouvant être chaîné

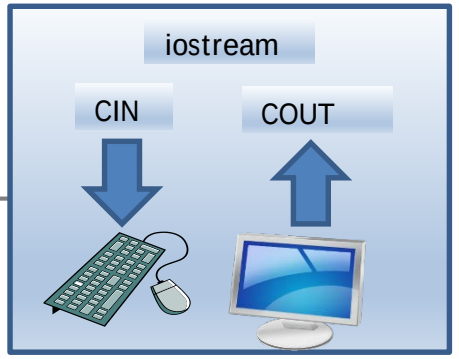


# E/S C++ - manipulateur des flux de sortie




- Les manipulateurs permettent de formater nos affichages
- Un manipulateur de flux est une instruction permettant de modifier le comportement du flux
  (Ex.: afficher ou non les nombres après la virgule ou réaliser un alignement en colonne)
- Il existe deux syntaxes pour utiliser un manipulateur:
  1. Appel d'une fonction :
     *nom_du_manipulateur(nom_du_flux);*
  2. Surcharge de l'opérateur << pour les flux :
     *nom_du_flux* **<<** *nom_du_manipulateur;*
- Les deux syntaxes sont tout à fait équivalentes

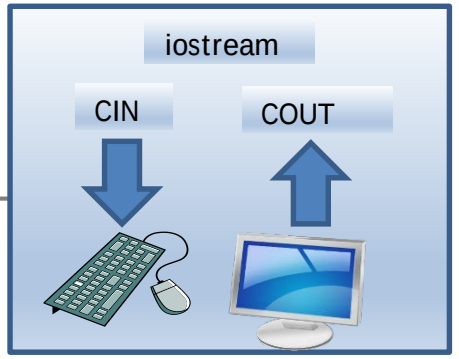
# E/S C++ - manipulateurs des flux de sortie




## Manipulateurs de sortie sans paramètre

| Manipulateur | Fonction (manipulateur sans paramètre) |
|---|---|
| endl | Envoie un caractère de retour à la ligne sur le flux et synchronise le tampon par un appel à la méthode flush. |
| ends | Envoie un caractère nul terminal de fin de ligne sur le flux. |
| flush | Synchronise le tampon utilisé par le flux par un appelle à la méthode flush. |
| boolalpha | Active le formatage des booléens sous forme textuelle. |
| noboolalpha | Désactive le formatage textuel des booléens. |
| hex | Formate les nombres en base 16. |
| oct | Formate les nombres en base 8. |
| dec | Formate les nombres en base 10. |
| fixed | Utilise la notation en virgule fixe pour les nombres à virgule. |
| scientific | Utilise la notation en virgule flottante pour les nombres à virgule. |
| left | Aligne les résultats à gauche. |
| right | Aligne les résultats à droite. |
| showbase | Indique la base de numérotation utilisée. |
| noshowbase | N'indique pas la base de numérotation utilisée. |
| showpoint | Utilise le séparateur de virgule dans les nombres à virgule, même si la partie fractionnaire est nulle. |
| noshowpoint | N'utilise le séparateur de virgule que si la partie fractionnaire des nombres à virgule flottante est significative. |
| showpos | Écrit systématiquement le signe des nombres, même s'ils sont positifs. |
| noshowpos | N'écrit le signe des nombres que s'ils sont négatifs. |
| uppercase | Écrit les exposants et les chiffres hexadécimaux en majuscule. |
| nouppercase | Écrit les exposants et les chiffres hexadécimaux en minuscule. |

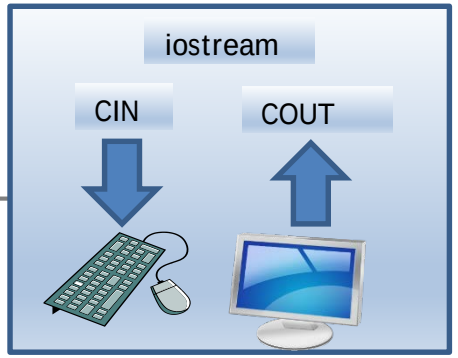
# E/S C++ - manipulateurs des flux de sortie




## Manipulateurs de sortie avec paramètre

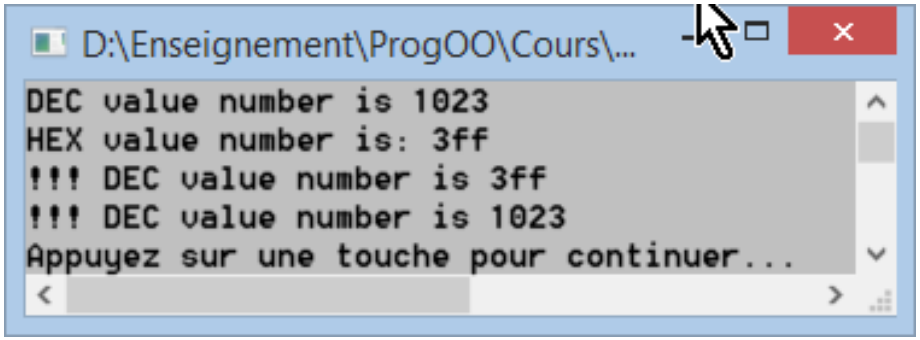
| Manipulateur | Fonction (manipulateur avec paramètre) |
|---|---|
| setbase(int base) | Permet de sélectionner la base de numérotation utilisée. Les valeurs admissibles sont 8, 10 et 16 respectivement pour la base octale, la base décimale et la base hexadécimale. |
| setprecision(int) | Permet de spécifier la précision (nombre de caractères significatifs) des nombres formatés. |
| setw(int) | Permet de spécifier la largeur minimale du champ dans lequel la donnée suivante sera écrite à la prochaine opération d'écriture sur le flux. |
| setfill(char_type) | Permet de spécifier le caractère de remplissage à utiliser lorsque la largeur des champs est inférieure à la largeur minimale spécifiée dans les options de formatage. |

# E/S C++ - manipulateurs des flux de sortie



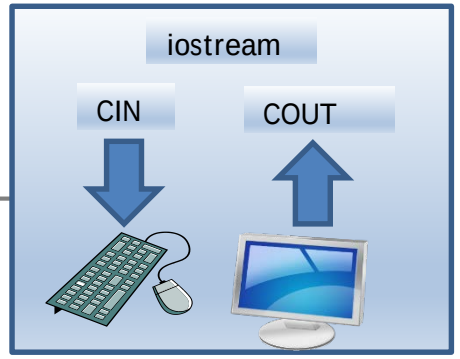

- Les manipulateurs conservent leur état jusqu'au prochain changement, sauf pour setw, qui revient à 0 après chaque opération
- Pour utiliser les manipulateurs, il faut inclure les fichiers <iostream> et avec certains <iomanip>

```
int number = 0x3FF;
cout << "DEC value number is " << number << '\n';
cout << "HEX value number is: " << hex << number << '\n';
cout << "!!! DEC value number is " << number << '\n';
dec(cout);
cout << "!!! DEC value number is " << number << '\n';
```



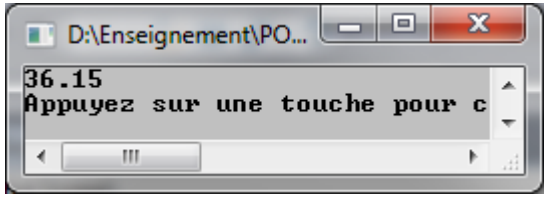
heig-vd

# E/S C++ - manipulateurs des flux de sortie

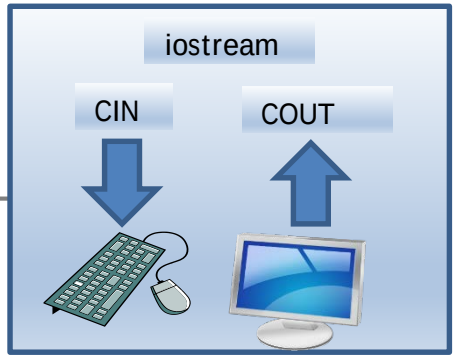



- Exemple de set précision:
  - Avec "setprecision(2)" le programme affichera que deux digits, à savoir 36
  - Pour préciser que le 2 correspond aux digits après la virgule, il faut ajouter "fixed"

```
#include <iomanip>
int main(void)
{
     double r = 36.145278;
     cout << fixed << setprecision(2) << r << endl;
     system("PAUSE");
     return EXIT_SUCCESS;
}
```



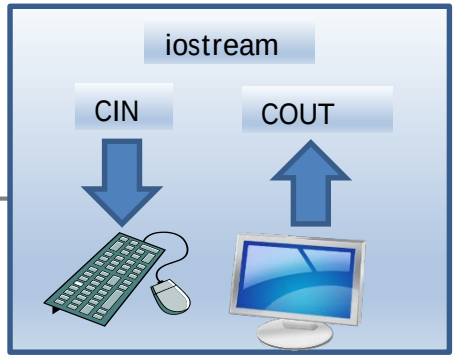
# E/S C++ - manipulateurs des flux d'entrée



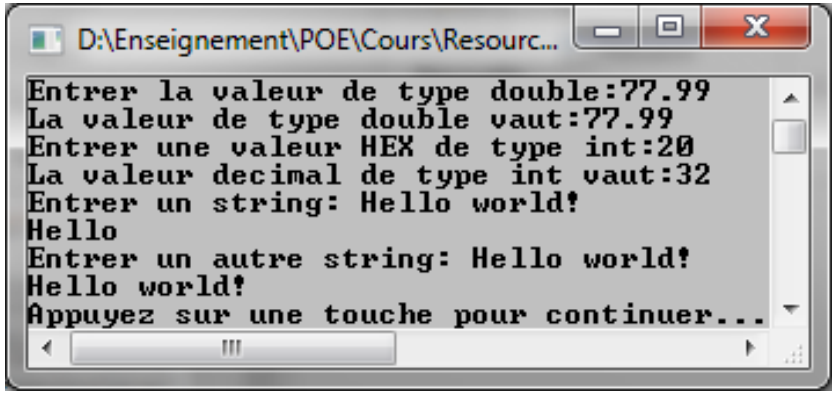

## Manipulateurs d'entrée

| Manipulateur | Fonction |
|---|---|
| boolalpha | Active l'interprétation des booléens sous forme de textuelle. |
| noboolalpha | Désactive l'interprétation des booléens sous forme textuelle. |
| hex | Utilise la base 16 pour l'interprétation des nombres entiers. |
| oct | Utilise la base 8 pour l'interprétation des nombres entiers. |
| dec | Utilise la base 10 pour l'interprétation des nombres entiers. |
| skipws | Ignore les espaces lors des entrées formatées. |
| noskipws | Conserve les espaces lors des entrées formatées. |
| ws | Supprime tous les espaces présents dans le flux d'entrée jusqu'au premier caractère non blanc. |

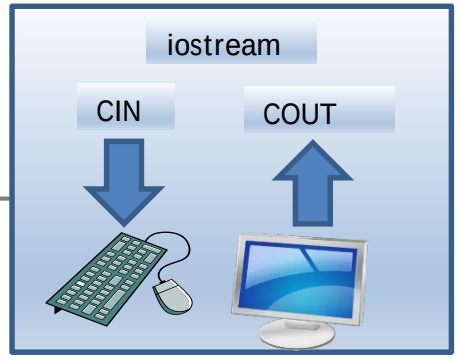
# E/S C++ - manipulateurs des flux d'entrée



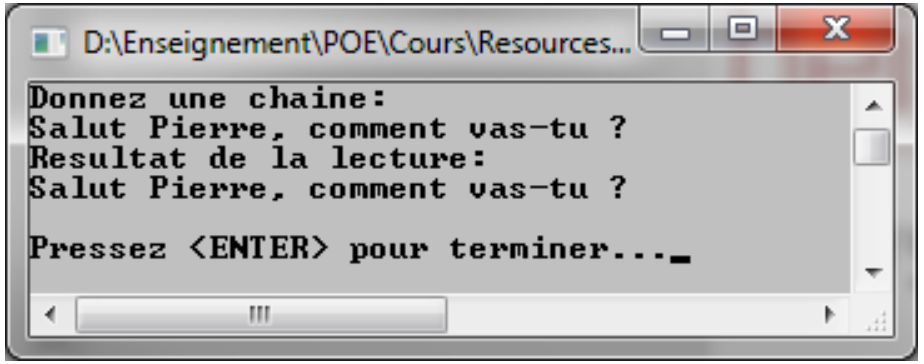

```
cout << "Entrer la valeur de type double:";
cin >> d;
cout << "La valeur de type double vaut:" << d << endl;
cout << "Entrer une valeur HEX de type int:";
cin >>hex >>i;
cout << "La valeur decimal de type int vaut:" << i << endl;
cout << "Entrer un string: ";
cin >> myString; //lit un string (s'arrête avec l'espace)
cin.sync();  // flush le buffer d'entrée !!! NOK depuis VS 2015
cin.ignore(MAX_FLUSH, '\n'); // flush le buffer d'entrée pour VS 2015
cout << myString << endl;
cout << "Entrer un autre string: ";
getline(cin, myString); // lit toute une ligne
cout << myString << endl;
```



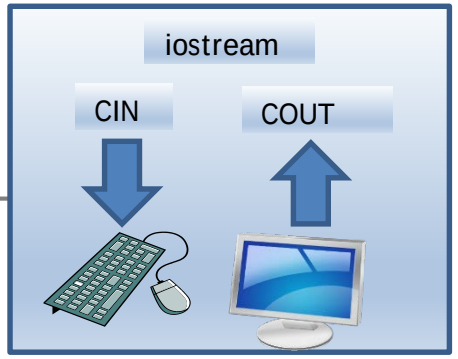
# E/S C++ - Lecture d'un string




```
int main(void)
{
    string s;
    cout << "Donnez une chaine:\n";
    getline(cin, s);
// cin >> s;    ne serait pas correct  car la lecture s'arrête à la
// rencontre du premier caractère blanc (espace, fin de ligne, etc)

    cout << "Resultat de la lecture:\n";
    cout << s << endl;
    cout << "\nPressez <ENTER> pour terminer...";
    cin.get();
    return EXIT_SUCCESS;
}
```



# E/S C++ - Vider le buffer d'entrée



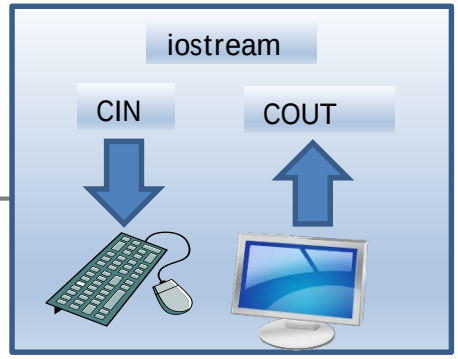

- Après un cin >> ... un '\n' demeure dans le tampon (voir plusieurs caractères si par exemple on entre un string après la valeur attendue)
- **Comme avec le langage C, il faut vider le tampon avant une nouvelle lecture**

- Par exemple, pour pouvoir effectuer un getline juste après un cin, il faut impérativement de vider le tampon (effacer le '\n')

- Pour ce faire, 2 possibilités:

```
1. while (cin.get() != '\n'); // vide le buffer d'entrée
2. cin.ignore(MAX_FLUSH, '\n'); // vide le buffer d'entrée

!!! cin.sync();  // → NOK depuis VS 2015
```

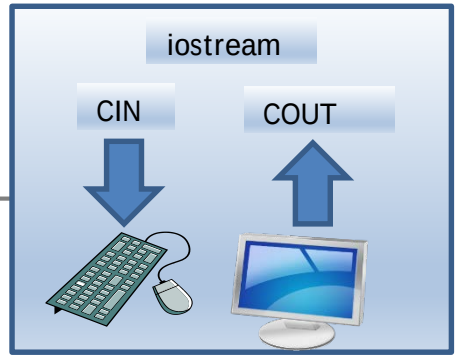
# E/S C++ - cin.ignore



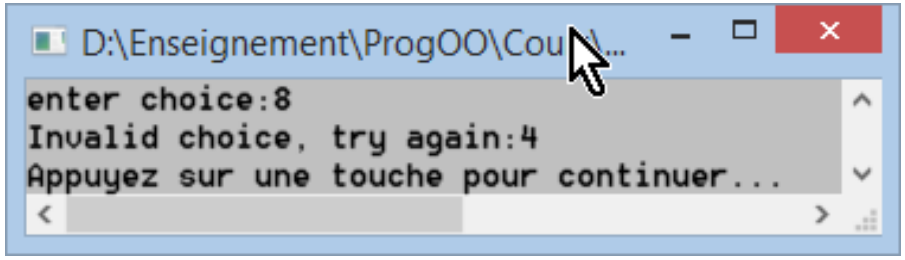

- Prototype de cin.ignore():
  - istream& ignore (streamsize n = 1, int delim = EOF);

- Exemples:
  - ```
    // ignore tous les caractères jusqu'à la fin de la ligne
    cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    ```
  - ```
    // ignore tous les caractères jusqu'à la fin du fichier
    cin.ignore(std::numeric_limits<std::streamsize>::max())
    ```

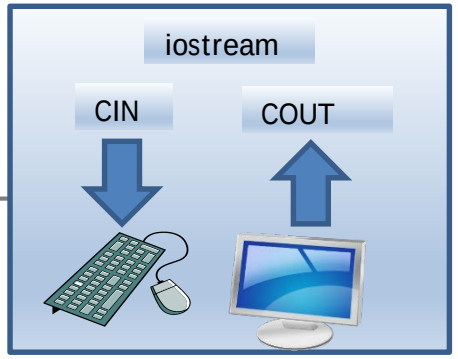
# E/S C++ - Tester si valeur valide



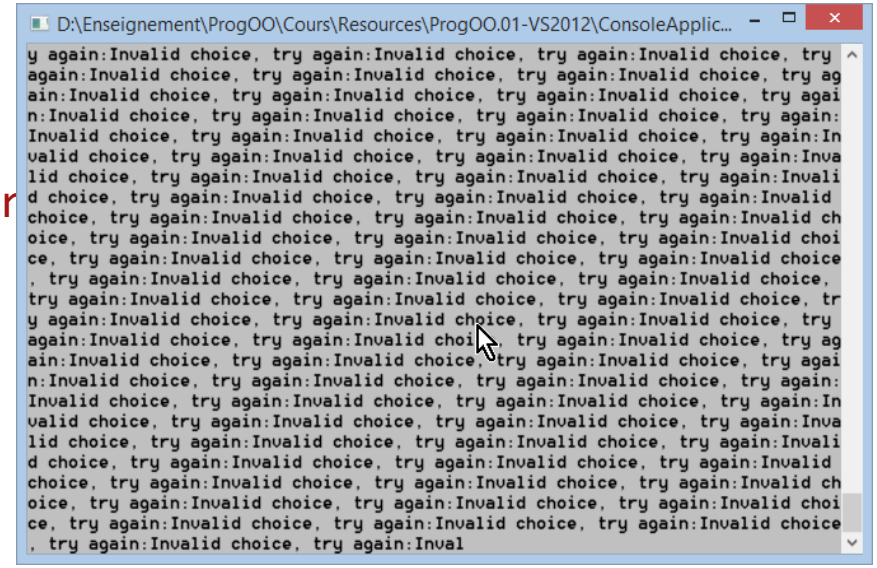

```
int x;
cout << "Entrer une valeur entre 1 et 4: ";
cin >> x;
while (x < 1 || x > 4)
{
   cout << "Invalid choice, try again:";

   if (cin.fail())  // test si iostream erreur
   {
      cin.clear(); // efface l'erreur de iostream
      cin.ignore(MAX_FLUSH, '\n'); // flush buffer d'entrée
   }
   cout << "Re-Entrer une valeur entre 1 et 4: ";
   cin >> x;
}
```



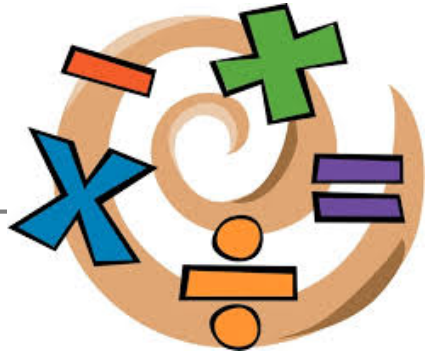
# E/S C++ - Tester si valeur valide




```
int x;
cout << "Entrer une valeur entre 1 et 4: ";
cin >> x;
while (x < 1 || x > 4)
{
    cout << "Invalid choice, try again:";
    //if (cin.fail())  // test si iostream erreur
    //{
    //cin.clear(); // efface l'erreur
    //cin.ignore(MAX_FLUSH, '\n'); //
    //}
    cout << "Re-Entrer une valeur entr
    cin >> x;
}
```



# Les types et opérateurs du C++



- Tous les types du C se retrouvent en C++
  - (int, unsigned int, short int, float, double, char, etc)
- Tous les opérateurs du C se retrouvent en C++
  - Affectation (=, +=, *=, etc)
  - Arithmétiques (+, -, *, /, %)
  - Relationnels (== !=, <, >, etc)
  - Logiques (&&, ||, !)
  - Décalage (>>, <<)
  - Binaires (&, |, ^)

# Les types supplémentaires du C++ - bool


- C++ introduit un type pour les conditions logiques (comme C99)
- Son utilisation est recommandée, mais pas obligatoire

```
bool valide;
valide = true;
valide = false;
if (valide)
    . . .
```

```
bool test = false;
cout << boolalpha << test << endl;
```

# Les types supplémentaires du C++ - string


- Une variable "string" se déclare ainsi:

```
string maChaine;
```

- L'initialisation se fait par des **constructeurs**

```
string maChaine ("Salut les copains");
```

- Crée et initialise "maChaine" à "Salut les copains"

```
string maChaine ("Salut les copains ", 5);
```

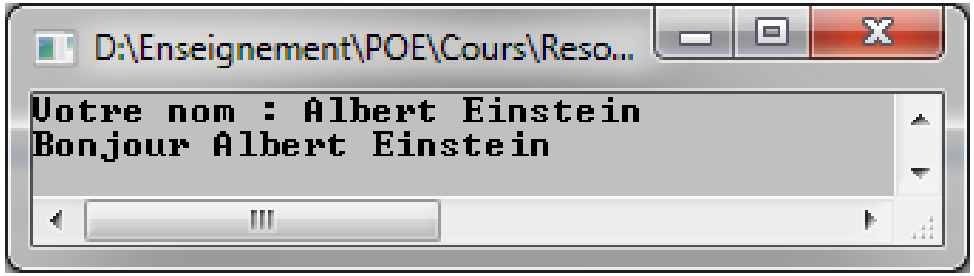
- Initialiser **maChaine** avec les **5 premiers caractères** de la chaîne (donc avec "Salut")

# Les types supplémentaires du C++ - string


- Il faut mettre (**#include <string>**)
  pour utiliser les string
- Ne pas confondre avec (**#include <cstring>**)
  à utiliser pour les fonctions strcpy, strstr, etc (bibliothèque du C)

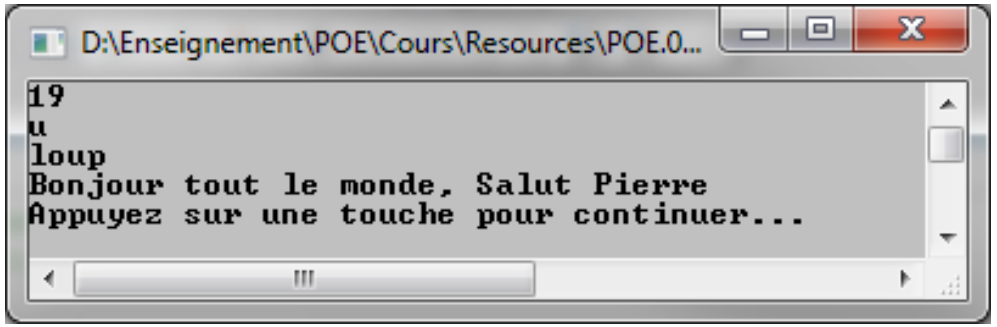
- Exemple

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

int main()
{
    string nom;
    cout << "Votre nom : ";
    // cin >> nom; : ne saisit que le 1er mot. Utiliser getline
    getline(cin, nom);
    cout << "Bonjour " << nom << endl;
    char c;
    cout << "Taper un caractere pour sortir" << endl;
    cin >> c; // remplace le _getch() ou system("PAUSE")
}
```



# string - quelques opérations


```
string chaine1, chaine2;
string chaine3 = "Bonjour tout le monde, ";
string chaine4("Salut Pierre");
string chaine5;

chaine1 = "Le loup et l'agneau";
// longueur de la chaine:
cout << chaine1.length() << endl;
// le caractère en position 5:
cout << chaine1.at(5) << endl; // aussi possible avec "chaine1[5]"
// sous-chaine de 4 caractères à partir de la position 3
chaine2 = chaine1.substr(3, 4);
cout << chaine2 << endl;
chaine5 = chaine3 + chaine4; // opération de concaténation
cout << chaine5 << endl;
```



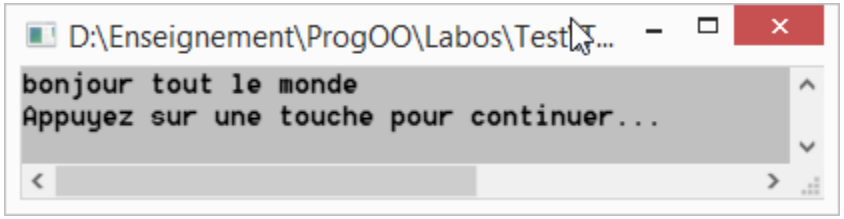
# String – Passage de paramètres aux fonctions


- Le passage d'un string en paramètre à une fonction est similaire aux passages des types simple (int, float, etc.)

- On peut passer un string:
  - **Par valeur**: void removeComment(string str);
    L'objet string est recopié et à la sortie de la fonction, l'objet initial n'est pas modifié
  - **Par référence**: void removeComment(string &str);
    L'objet string original est utilisé dans la fonction et à la sortie de la fonction, l'objet a les valeurs modifiées (pas de copie de l'objet)
  - **Par référence avec const**: void removeComment(const string &str);
    Pour un code efficace, mais en étant sûr que le string n'est pas modifé dans la fonction
  - **Par pointeur:** void removeComment(string *str);
    Solution conseillé par B. Stroustrup, mais plus délicate (il faut passer l'adresse du string à l'appel de la fonction)

# String – Passage de paramètres aux fonctions


```
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

void test(string *phrase)
{
  for (int i = 0; i < phrase->length(); ++i)
  {
    char caract = phrase->at(i);
    cout << caract;
  }
  cout << endl;
}
```




```
int main(void)
{
  string phrase = "bonjour tout le monde";

  test(&phrase); // passage par adresse
  system("PAUSE");
}
```

# String – c_str TODO


```
string nomFichierL;
FILE *fichier = NULL;
int nbrMots;
double coor[4];
Ligne *ligne;
char ligneTexte[100];

if (nomFichier == "")
{
        cout << "Nom du fichier sans ext. ? ";
        cin >> nomFichier;
}

nomFichierL = nomFichier + "L.txt";
fichier = fopen(nomFichierL.c_str(), "r");
```

# Les types supplémentaires du C++ - reference

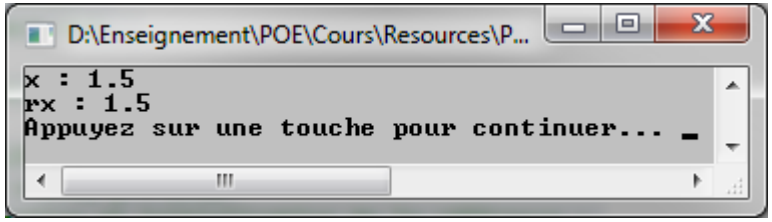

- Définition
  - Un type référence est en fait un pointeur
  - Utilisable SANS la syntaxe fastidieuse des pointeurs du C
  - Une **référence** est un **synonyme** « alias » **d'un autre objet existant**
- Particularités
  - Une variable de type référence DOIT être initialisée
  - L'initialisation indique quelle variable est référencée
  - Il n'est ensuite plus possible de modifier la référence
- Syntaxe

```
type &nom_reference = variable_referencee;
```

# Le type "reference" en C++ - Exemple

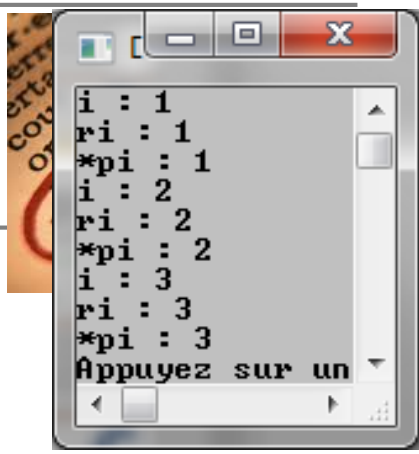

- Exemple

```
double x = 0.0, y = 1.5;
// déclaration et initialisation de la référence :
double &rx = x;

// affectation de la variable référencée
rx = y;
cout << "x :" << x << endl;
cout << "rx :" << rx << endl;
```

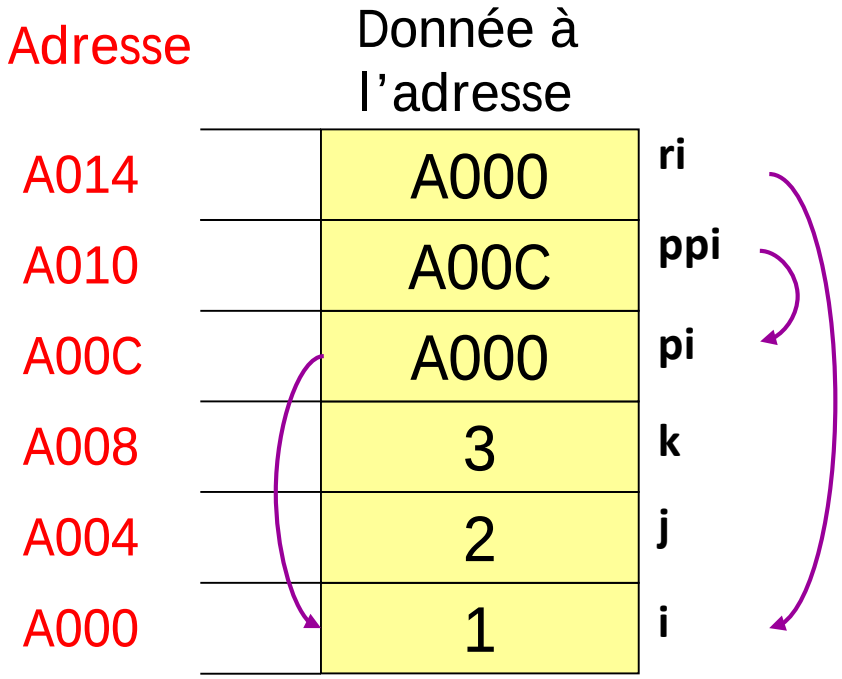



- Comportement
  - **A l'initialisation, la référence reçoit l'adresse d'une variable**
  - **Lors de toutes les affectations ultérieures, c'est la variable référencée qui est modifiée**
  - **La référence désigne donc toujours la même variable pendant toute sa durée de vie**

# Types "reference" et "pointeur" en C++




```
int i = 1, j = 2, k = 3;
int* pi;  // pi contient n'importe quoi
pi = &i;  // pi contient l'adresse où se trouve i
int **ppi = &pi; // ppi contient A00C
int &ri = i; // ri contient l'adresse où se trouve i

cout << "i : " << i << endl;
cout << "ri : " << ri << endl;
cout << "*pi : " << *pi << endl; // déréférencement
ri = j;
cout << "i : " << i << endl;
cout << "ri : " << ri << endl;
cout << "*pi : " << *pi << endl; // déréférencement
*pi = k;
cout << "i : " << i << endl;
cout << "ri : " << ri << endl;
cout << "*pi : " << *pi << endl; // déréférencement
```




**Une référence est comme un pointeur qu'il ne faut pas déréférencer pour accéder à sa valeur**

***Adresse codée sur 16 bits (short), mais sur nos PC, l'adresse est codé sur 32 bits/64 bits**

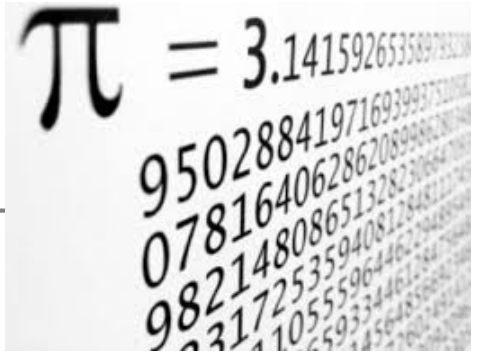
# Les variables en C++

| Variables | | |
|---|---|---|
| **Adresses** | **Valeurs** | **Nom** |
| A00C | A000 | pti1 |
| A008 | 3 | i3 |
| A004 | 2 | i2 |
| A000 | 1 | i1 |

- En C++, les variables **se déclarent** comme en C
  - Nous pouvons les initialiser avec la syntaxe du C
    - TYPE NOM = VALEUR;
  - Mais nous pouvons aussi les initialiser de cette manière:
    - TYPE NOM (VALEUR);
- En C++, on peut déclarer les variables partout où on peut mettre une instruction
  (plus nécessairement avant les instructions)

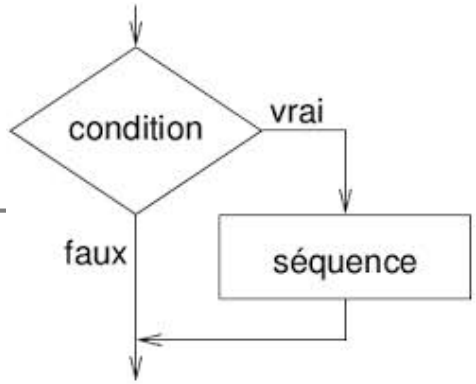


# Les constantes en C++




- Constantes symboliques
  - const unsigned int NOMBRE_ELEVE = 18;
- Constantes énumérées
  - enum COULEURS {ROUGE = 55, BLEU = 77, VERT, JAUNE}
- Constantes avec #define (à éviter car déclaré comme obsolète avec C++)
  - #define NOMBRE_ELEVE 18
- Constantes littérales (à éviter comme en C)
  - unsigned int nombreEleve = 18;

# Structures de contrôle en C++




- Tous les structures de contrôles du C se retrouvent en C++
  - (séquence, if, switch, while, do while, for, *goto*)

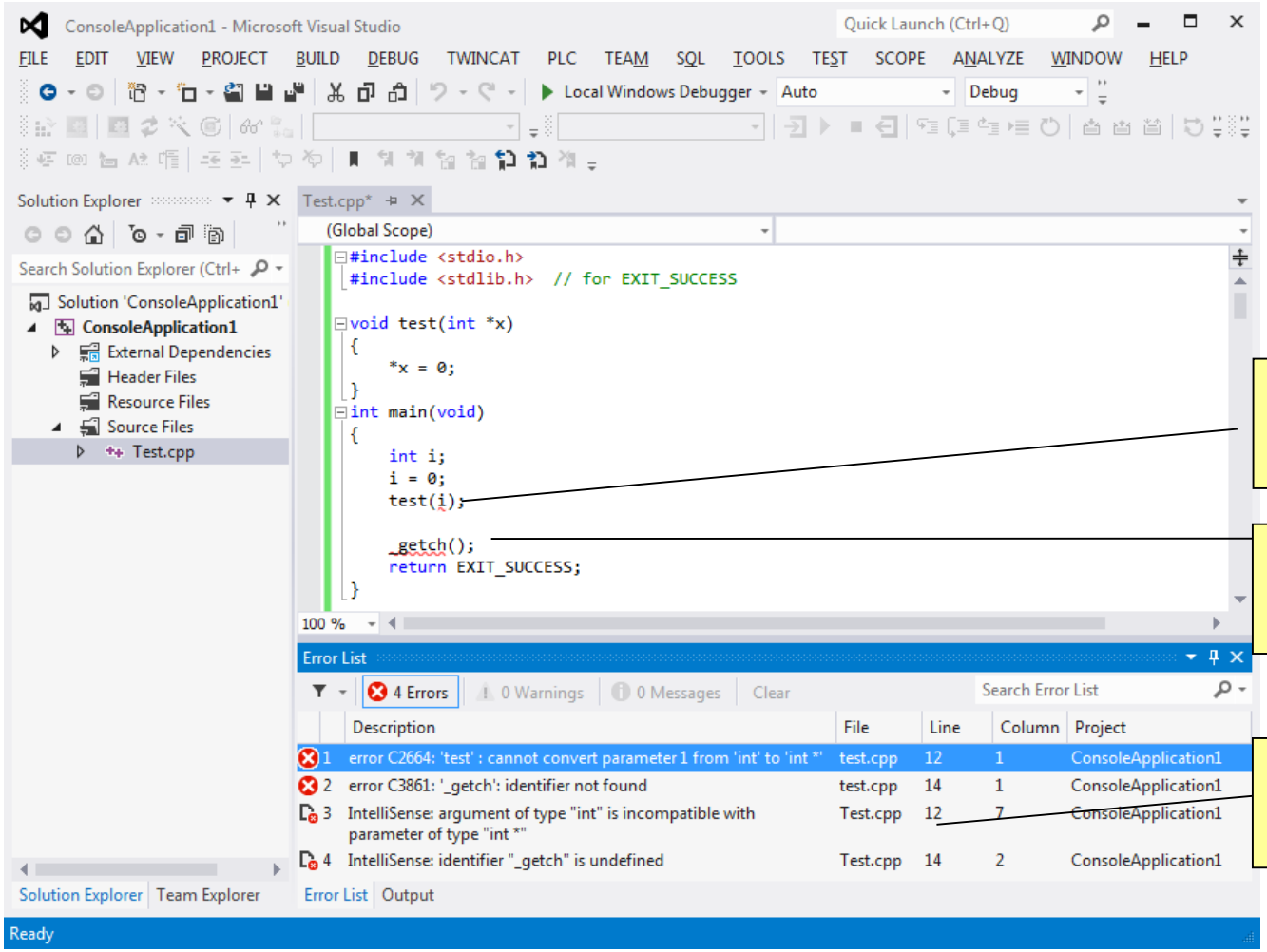
- L'index de boucle peut être déclarer dans la structure "for" (comme dans le C99)

```
for(int i = 0; i < 128; i = i + 1)
```

# Les fonctions C++ (non membres)


- Les fonctions non-membres (ou globales) sont les fonctions déclarées hors des classes
- Identiques aux fonctions en C avec en plus:
  1. Les paramètres par défaut
  2. La surcharge
- Le compilateur C++ est plus strict
  - Le prototype d'une fonction doit être défini avant son utilisation
  - Erreur systématique générée en cas contraire
  - Pas de supposition sur le type du résultat et des paramètres
  - Contrôle des types des paramètres lors d'appels de fonctions
  - L'exécution d'une fonction doit toujours se terminer par un return (si pas "void")

# Les fonctions C++ - Erreurs de compilation




Type du paramètre incorrect

Fonction sans prototype

En C++, erreurs de compilation générées.

# Les fonctions C++ - Paramètres par défaut

- Les paramètres formels d'une fonction peuvent avoir des valeurs par défaut. Exemple:

```
void trier(void *table, int nbr, int taille = sizeof(void *),
           bool croissant = true);
```

- Lors de l'appel d'une telle fonction, les paramètres effectifs correspondants **peuvent alors être omis**

```
trier(t, n, sizeof(int), false); // tous les paramètres
trier(t, n, sizeof(int)); // croissant = true
trier(t, n); // taille = sizeof(void *), croissant = true
```

# Les fonctions C++ - Paramètres par défaut

- Règles pour paramètres par défaut
  - Lorsqu'un paramètre a une valeur par défaut, tous les paramètres qui suivent doivent également en avoir une.
  - Si le paramètre est fourni au moment de l'appel, c'est la valeur fournie qui est utilisée.
  - Sinon, c'est la valeur par défaut.
  - Une fonction avec des paramètres par défaut peut donc être appelée avec un nombre variable de paramètres.

# Les fonctions C++ - La surcharge

- C++ permet de créer plusieurs fonctions portant le même nom. C'est ce qu'on appelle "surcharge de fonction"

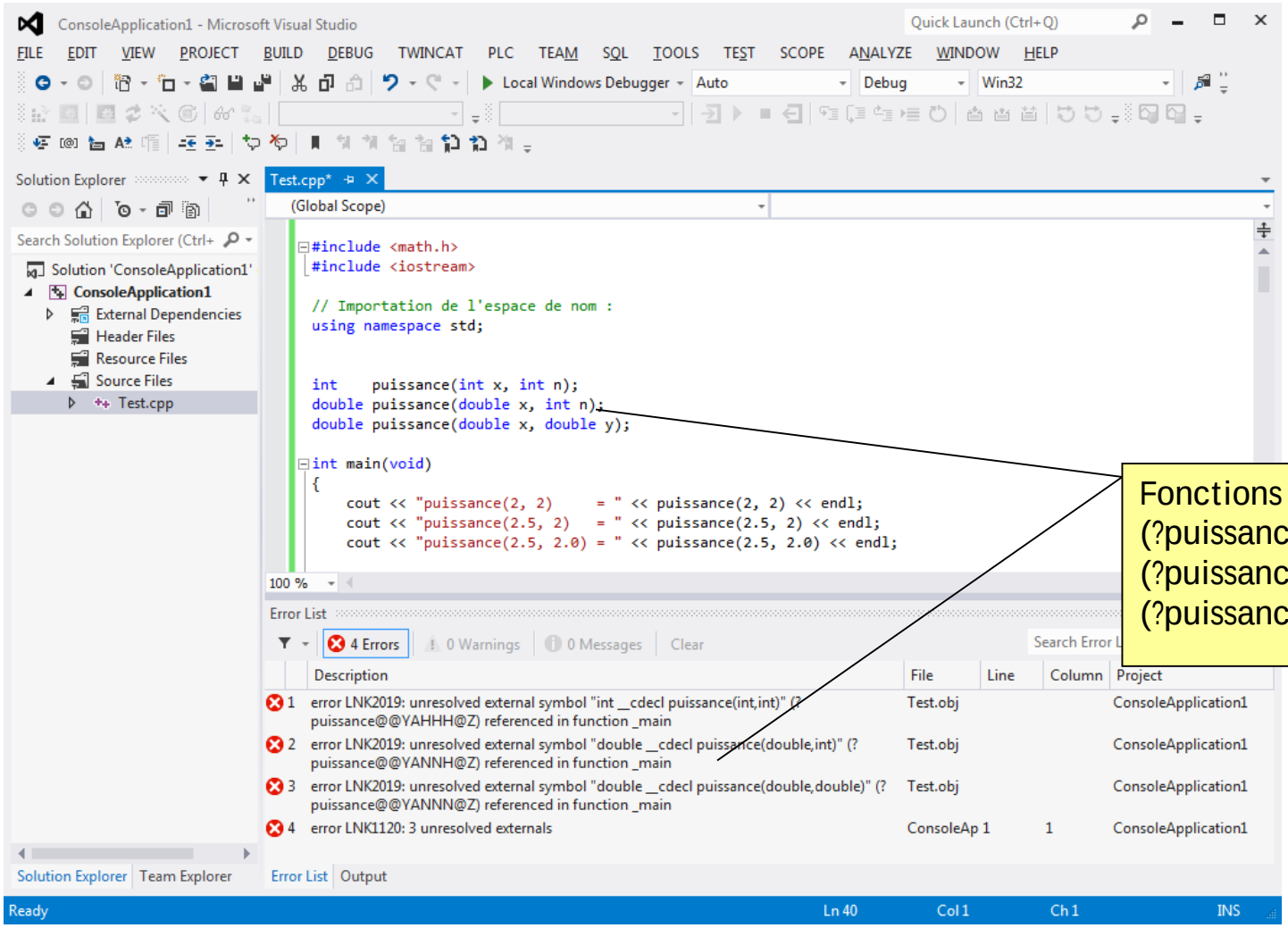
- La signature des fonctions doit différer par le type et/ou le nombre des paramètres. Exemple:

```
int    puissance(int x, int n);
double puissance(double x, int n);
double puissance(double x, double y);
```

# Les fonctions C++ - La surcharge

- A l'issue de la compilation
  - Le code objet contient plusieurs fonctions puissance.
  - Comment le lieur (link) peut il retrouver la bonne ?
- Dans le code généré par le C++
  - Les noms de fonctions sont suffixés par un encodage de la signature.
  - 2 fonctions de signature différente ont ainsi un nom différent.

# Les fonctions C++ - La surcharge

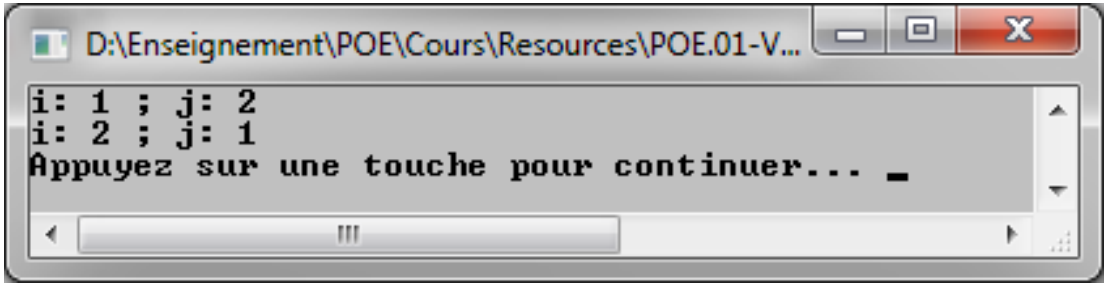


# Les fonctions C++ - Passage de paramètres par référence

- Une référence est en fait un pointeur
- Permet donc de réaliser le passage de paramètre par adresse, ou en autre terme, permettre de donner aux fonctions des paramètres modifiables
- SANS la syntaxe lourde des pointeurs
- Avec les références, C++, offre une solution élégante pour le passage de paramètres par variable (par référence)

# Les fonctions C++ - Exemple
# Passage de paramètres par référence

```
void echanger(int &a, int &b)
{
    int temporaire = a;
    a = b;
    b = temporaire;
}

int main(void)
{
    int i, j;
    i = 1;    j = 2;
    cout << "i: " << i << " ; j: " << j << endl;
    echanger(i, j);
    cout << "i: " << i << " ; j: " << j << endl;
}
```

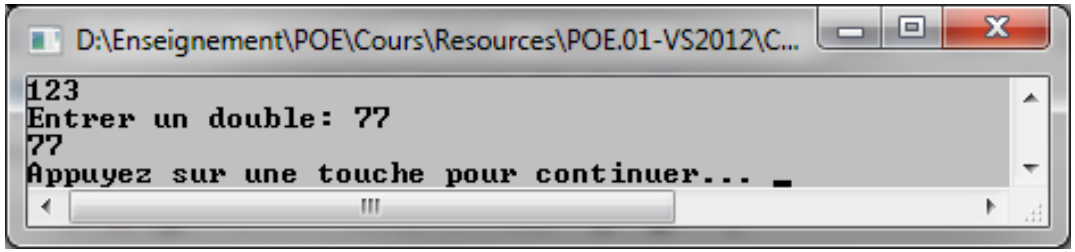


# Les fonctions C++ - Renvoyant une référence

- Une fonction peut retourner une référence (donc une adresse)
  - On peut affecter la variable référencée retournée
  - On peut ainsi placer un appel de fonction à gauche d'une affectation !

- Exemple

```
double tableau[100];

double & element(int i)
{
    return tableau[i];
}
int main(void)
{
   element(5) = 123; // appel de fonction à gauche!! avec affectation val retournée
   cout << element(5) << endl;
   cout << "Entrer un double: ";
   cin >> element(10);
   cout << element(10) << endl;
}
```



# Attention au signe &

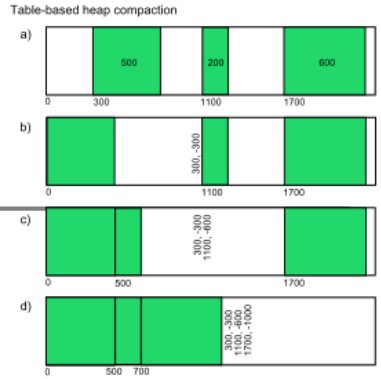
- ***Attention:*** On notera que l'opérateur & (à un argument) a une signification très différente selon le contexte dans lequel il apparait!

- Employé dans une déclaration, comme dans:
  - **int** &r = x;
  - il sert à indiquer un type référence : « r est une référence sur un int »

- Employé ailleurs que dans une déclaration, il indique l'opération « obtention de l'adresse », comme dans l'expression suivante:
  - p = &x;
  - Il affecte l'adresse de x à p

- enfin, on doit se souvenir qu'il y a en C++, comme en C, un opérateur & binaire (à deux arguments) qui exprime la conjonction bit-à-bit de deux mots-machine

# Allocation dynamique de mémoire - Allocation



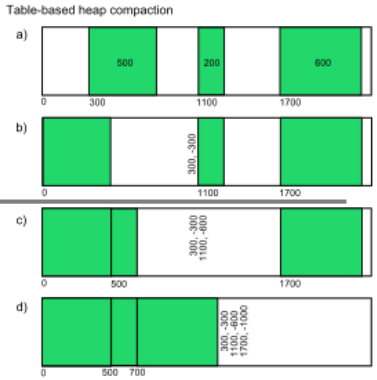

- C++ offre un opérateur pour l'allocation dynamique
  - Opérateur new
  - Syntaxe et utilisation beaucoup plus claires que les malloc du C
- Syntaxe

```
TYPE *variable;
variable = new TYPE;
```

- Exemple – allocation d'une cellule pour une file chainée

```
struct CELLULE_FILE_CHAINEE *cellule;

cellule = new CELLULE_FILE_CHAINEE;
cellule->element = 10.5;
```

# Allocation dynamique de mémoire - Libération

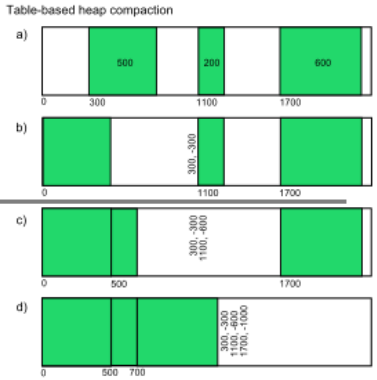



- C++ offre également un opérateur pour la libération
  - Opérateur delete

- Syntaxe

```
TYPE *variable;
variable = new TYPE;
. . .
delete variable;
```

# Allocation dynamique de mémoire - Tableaux




- L'opérateur new permet d'allouer un tableau d'éléments

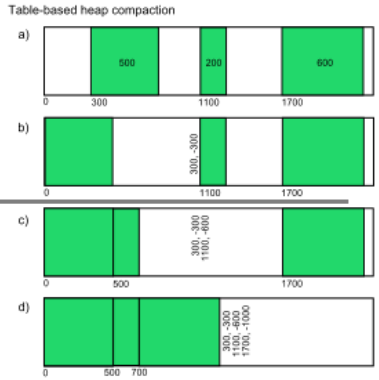
```
tableau = new type[taille];
```

- Syntaxe de l'opérateur delete pour les tableaux

```
delete []tableau;
```

- Allocation dynamique du C++
  - Plus facile à utiliser
  - Plus lisible
  - A préférer pour tout développement en C++
  - A utiliser impérativement dès qu'on alloue des objets

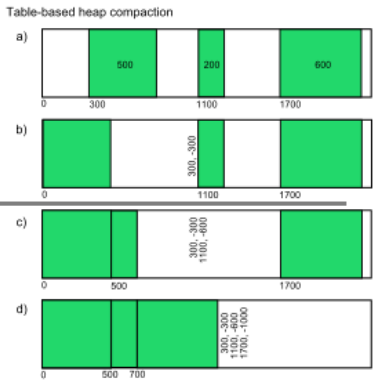
# Allocation dynamique de mémoire - Tableaux




- Exemple d'allocation/libération dynamique d'un tableau de pointeur

```
const int n; // taille du tableau
const int p; // taille de la phrase
char **tab=new char*[n];
for(int i = 0; i<n; i++)
        tab[i]=new char[p];
 …
for(int i = 0; i<n; i++)
        delete []tab[i];
delete tab;
```

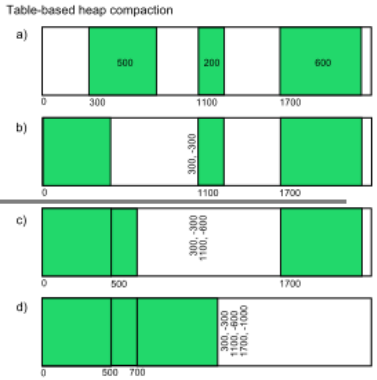
# Allocation dynamique de mémoire – version C




- La bibliothèque "stdlib" permet d'utiliser les fonctions d'allocation du C
- Elle met à disposition: malloc, calloc, realloc et free
- Mais il ne faut pas mélanger les opérateurs C et C++

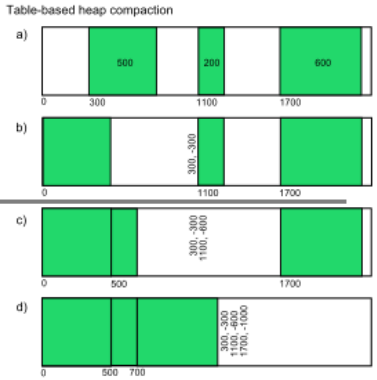
| | allocation | libération |
|---|---|---|
| C | `malloc`<br>`calloc`<br>`realloc` | `free` |
| C++ | `new`<br>`new[]` | `delete`<br>`delete[]` |

# Allocation dynamique de mémoire – version C




- *realloc* n'existe plus en C++
- Pour "agrandir/réduire" un tableau il faut:
  - réallouer un nouveau tableau de la taille voulue
  - copier les anciennes valeurs vers ce nouveau tableau
  - détruire l'ancien tableau

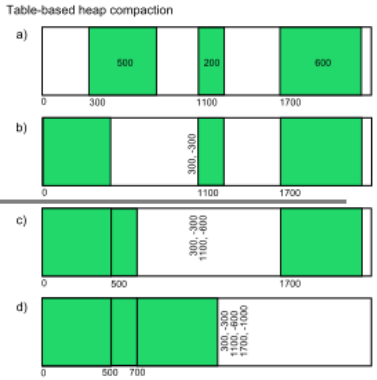
# Allocation dynamique de mémoire - Erreur




- Que se passe-t-il si la **mémoire** disponible est **insuffisante** et ne **permet pas la création** de la variable dynamique ?
  - L'exception « **bad_alloc** » sera levée (a voir plus tard dans cours)
  - Ou demander de **ne pas avoir d'exception** propagée avec:
    *p_var* = **new**(nothrow) *type*;

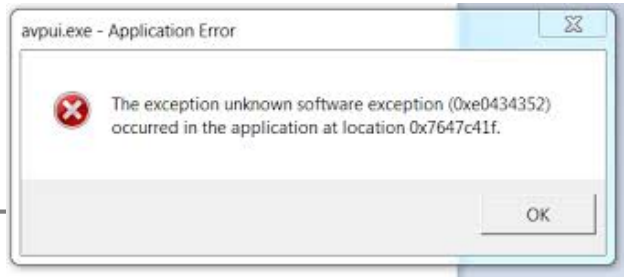
    Et tester si l'adresse est différent de NULL
    ```
    if (pt != NULL) …
    ```

# Allocation dynamique de mémoire - Exemple




```
int main(void)
{
    double * tableau;
    int * ptentier;

    int i, taille;
    ptentier = new int(10); //Allocation pour 1 entier contenant la valeur 10
    cout << "Taille:";
    cin >> taille;

    tableau = new double[taille]; // Allocation dynamique du tableau
    for (i = 0; i < taille; i++)
    {
        cout << "Element " << i + 1 << " : ";
        cin >> tableau[i];
    }
    // Libération du tableau
    delete []tableau;
    delete ptentier;
    return EXIT_SUCCESS;
}
```

# La gestion des erreurs exceptionnelles




```
int ListeDouble::Ajouter(double x)
{
    int indice;
    if (nombre_elements < CAPACITE)
    {
        indice = nombre_elements;
        tableau[indice] = x;
        nombre_elements++;
    }
    else
        indice = INDICE_INVALIDE;
    return indice;
}
```

```
int Filtre::Echantilloner(double amplitude)
{
    if (liste.Ajouter(amplitude) != INDICE_INVALIDE)
        return 1;
    else
    {
        printf("Erreur");
        return 0;
    }
}
```

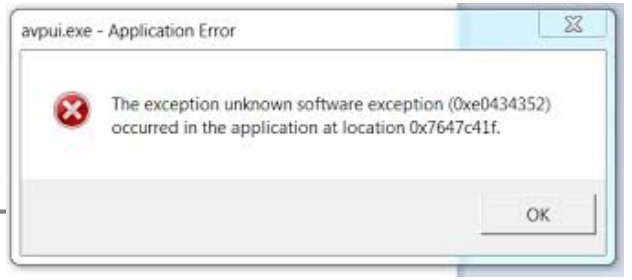
```
double FiltreMedian(double valeurs[], int nombre)
{
    int i;
    Filtre f;

    for (i = 0; i < nombre_valeurs; i++)
        if (!f.Echantilloner(valeurs[i]))
            return AMPLITUDE_INVALIDE;
    return f.CalculerValeurMediane();
}
```

Le code produit pour gérer les erreurs exceptionnelles représente 50 %

Beaucoup d'effort pour propager l'erreur « liste pleine » vers les fonctions de niveau supérieur.

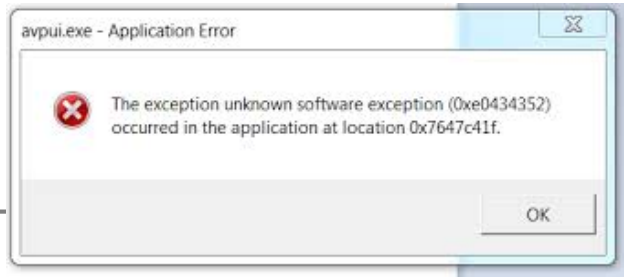
# Les exceptions en C++

## Nouvelle approche de traitement d'erreurs




- Lors de la détections d'une erreur
  - On « lève » une exception
  - Une exception est un signal logiciel, accompagné d'une donnée
  - La donnée peut être de n'importe quel type
    - Souvent un objet, de classe (ou sous classe) « exception »
- Le traitement en cours est interrompu
- L'exception se propage à travers les fonctions appelantes
  - Sans avoir aucun code à écrire pour cela
- Jusqu'à ce qu'elle soit capturée
  - Au moment de la capture, la donnée associée est récupérable
- Si elle n'est pas traitée
  - En général : terminaison de l'application

# Les exceptions - Syntaxe



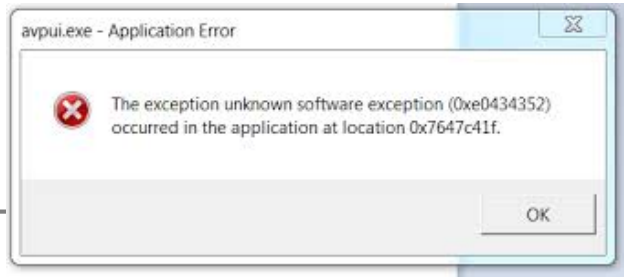

- Lever (forcer) une exception

```
throw exception("Liste pleine");
throw 1.5; // pas très usuel
```

- Capturer une exception

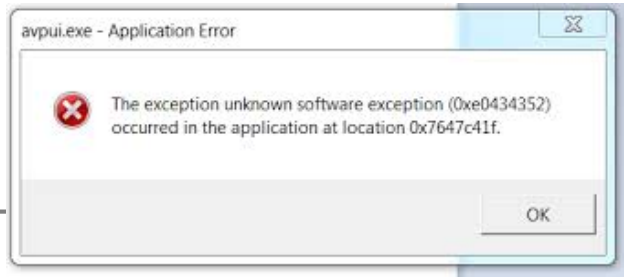
```
try
{
    // Placer ici le code pouvant générer une exception
    // que l'on veut capturer
}
// blocs de capture : un ou plusieurs selon les besoins
catch (exception & e)
{
    cout << e.what() << endl;
}
catch (...)  // ... signifie capturer tout
{
    cout << "Exception de type inattendu\n" << endl;
    throw; // redéclenchement de l'exception capturée
}
```

# Les exceptions - Syntaxe




```
try {
    // Code qui peut générer des exceptions
}
catch(type1 id1) {
    // prend en charge les exceptions de type1 (e.g. int)
}
catch(type2 id2) {
    // prend en charge les exceptions de type2 (e.g. type exception)
}
catch(type3 id3) {
    // Etc...
}
catch(…) {
    // prend en charge toutes les autres exceptions
}
// L'exécution continue ici
```

# La gestion des erreurs avec les exceptions




```
int ListeDouble::Ajouter(double x)
{
    int indice;
    if (nombre_elements < CAPACITE)
    {
        indice = nombre_elements;
        tableau[indice] = x;
        nombre_elements++;
    }
    else
        throw exception("Liste pleine");
    return indice;
}
```

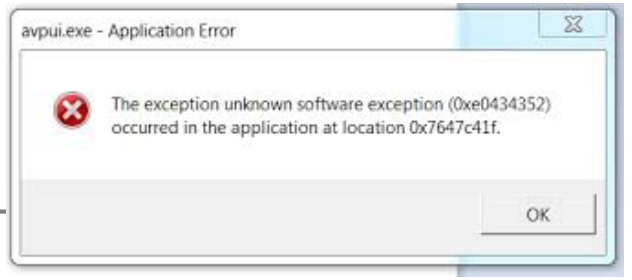
```
void Filtre::Echantilloner(double amplitude)
{
    liste.Ajouter(amplitude);
}
```

```
double FiltreMedian(double valeurs[], int nombre)
{
    int i;
    Filtre f;

    for (i = 0; i < nombre_valeurs; i++)
        f.Echantilloner(valeurs[i]);
    return f.CalculerValeurMediane();
}
```

Plus aucun code à écrire pour la propagation des erreurs !

Programmation beaucoup plus efficace.

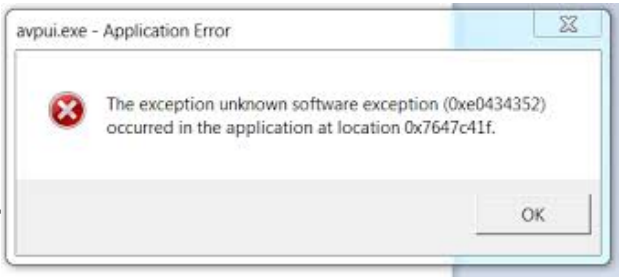
# La gestion des erreurs avec les exceptions




- Traitement de l'exception
  - Se fait seulement au niveau où l'on sait comment réagir.
  - Exemple : dans main, afficher simplement un message d'erreur.
- Exemple simple

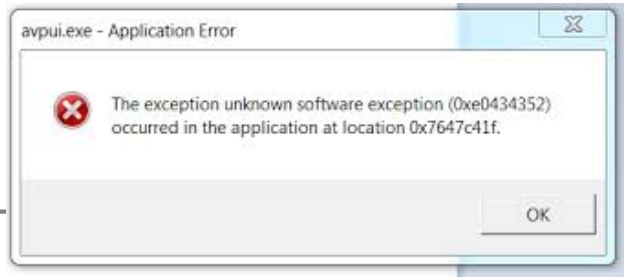
```
double acquisitions[100];
double valeur_filtree;
. . .

try
{
    valeur_filtree = filtre_median(acquisitions, 100);
}
catch (exception & e)
{
    cout << "Erreur durant le filtrage : " << e.what() << endl;
}
```

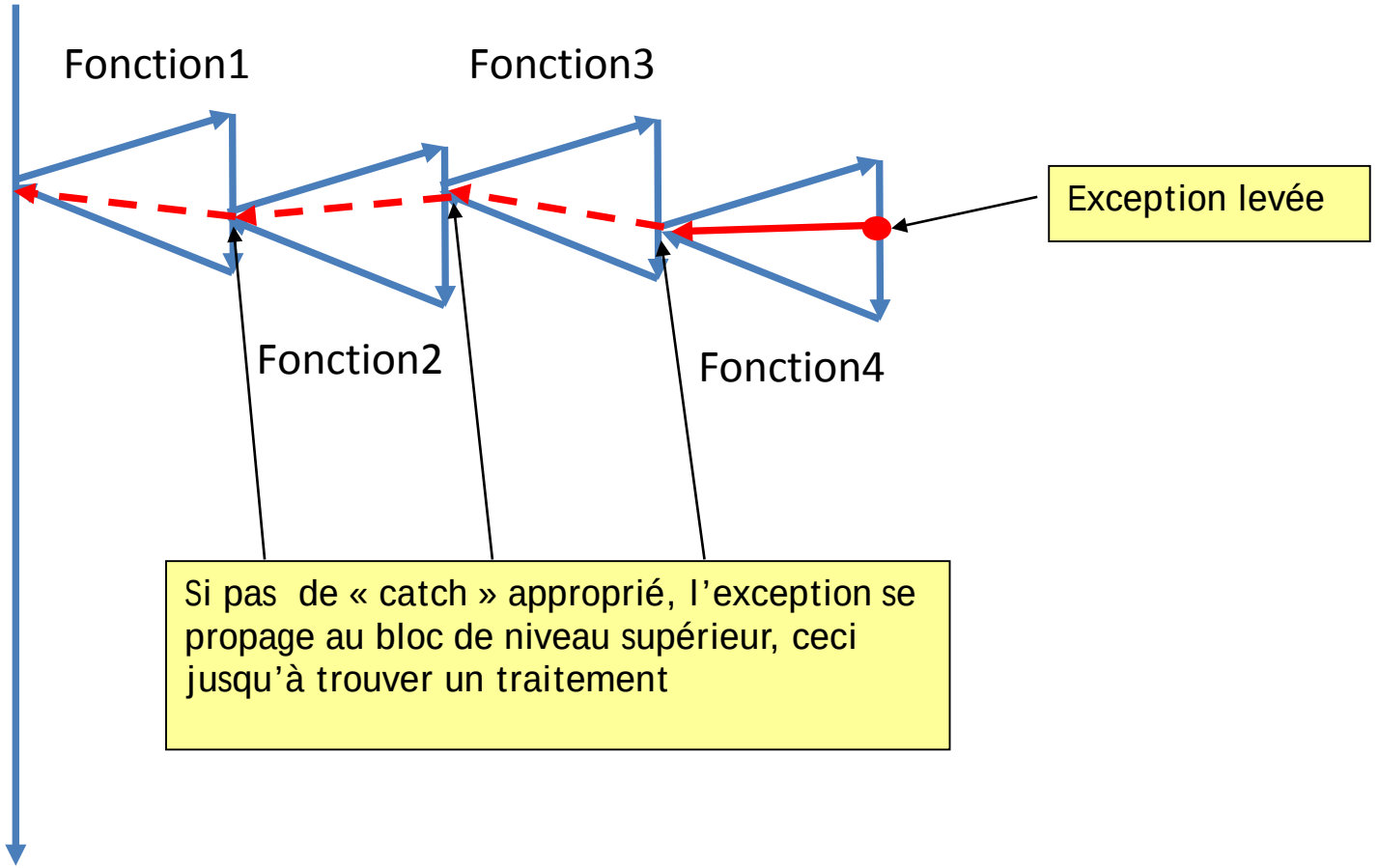
# La gestion des erreurs avec les exceptions



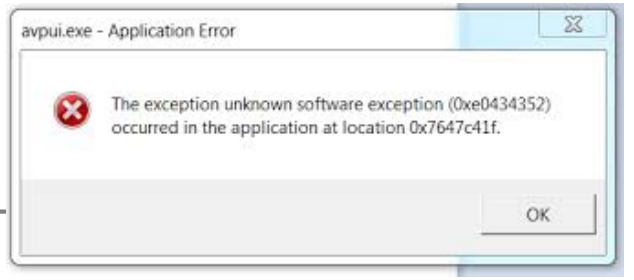

- Si aucune exception n'est levée dans le bloc « try », tout se passe comme si les blocs « catch » n'existaient pas
- Si une exception est levée, l'exécution est abandonnée pour se poursuivre par les instructions du bloc « catch » correspondant
- Après l'exécution des instructions du bloc « catch » choisi, l'exception n'existe plus et le programme se poursuit à l'instruction suivant le dernier bloc « catch » associé au « try » (c'est-à-dire, après le catch)
- Après les dernières instructions du bloc « try », le programme se poursuit normalement à la première instruction qui suit le dernier bloc « catch » associé
- Si aucun bloc « catch » ne comporte un paramètre approprié, l'exception se propage au bloc de niveau supérieur, ceci jusqu'à trouver un traitement

# Propagation des exceptions



# La gestion des erreurs avec les exceptions




- Recommandation
  - Utiliser la gestion par exception seulement pour les situations exceptionnelles
  - Nous ne pouvons pas redonner le contrôle à l'endroit où l'exception a été levée

  - Une boucle (englobant "try & catch") permet de reprendre à l'endroit désiré
  - Préférer des tests avec if pour les situations probables

# Mélanger des fichiers C et C++

- Problème lors du mélange de fichiers C et C++

Lib.h

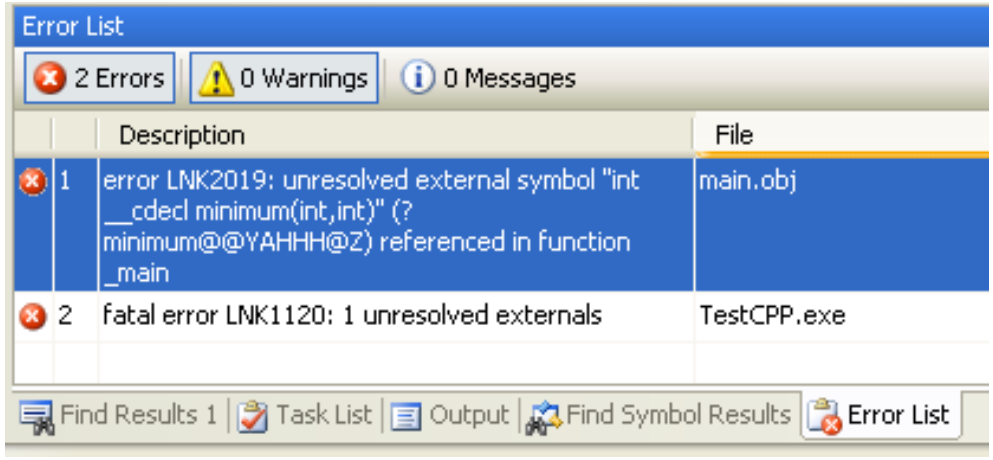
```
#ifndef __LIB_H_
#define __LIB_H_

int minimum(int a, int b);

#endif
```

Lib.c

```
#include "lib.h"

int minimum(int a, int b)
{
    return a < b ? a : b;
}
```

Main.cpp

```
#include "lib.h"

int main()
{
    int a;
    a = minimum(1, 2);
    return EXIT_SUCCESS;
}
```



# Mélanger des fichiers C et C++

Main.cpp

```
#include "lib.h"

int main(void)
{
    int a;
    a = minimum(1, 2);
    return EXIT_SUCCESS;
}
```

Lib.c

```
#include "lib.h"

int minimum(int a, int b)
{
    return a < b ? a : b;
}
```

Main.obj

_main, utilise ?minimum@@YAHHH@Z

Lib.obj

_minimum

Liaison impossible

# Mélanger des fichiers C et C++

- Solution
  - Il faut indiquer au compilateur C++ quelles fonctions utilisent les conventions de compilation du C plutôt que C++.
  - Syntaxe

```
extern "C"
{
    // déclarations à interpréter comme du C
}
```

- Exemple

**Main.cpp**

```
extern "C"
{
    #include "lib.h"
}

int main(void)
{
    int a;
    a = minimum(1, 2);
    return EXIT_SUCCESS;
}
```

heig-vd

# Vos questions


Exercices
Les bases du C++