ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ

Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. *Intro, concepts, 1 exemple*		1. *Organisation objet des données*
2. *Modélisation objet / UML*		2. *Diagrammes de classe UML*
3. C++ pratique 1		3. *C++ pratique, E/S, string, vector*
4. *C++ pratique 2*		4. *C++ pratique, type &, surcharge*
5. *Classes & C++ : bases*		5. *Date : une classe simple en C++*
6. *Classes & C++ : compléments*		6. *UML et C++, associations*
7. *Conteneurs & C++ : la STL*		7. *Gestion de collections complexes*
8. *Héritage / polymorphisme*		8. *Collections polymorphes*
9. *Modèles objets avancés*		9. *Modèle composite et graphismes*
10. *Exceptions, flots, fichiers ...*		10. *Persistance / fichiers / except.*
11. *Templates côté développeur*		11. *Développement de templates*
12. *Gestion mémoire / smarts ptrs*		12. *Soutenance de **projet** ...*

C++ *pratique 1*


Quincaillerie Moderne
OUTILLAGE
METAUX
ELECTRICITE
Articles pour
Cadeaux
BROSSERIE
DROGUERIE
CONSEIL
FOURNITURES
POUR
ARTISTES
PHOTOS
SUNLIGHT
SAVON
Persil
Corail
BRILLANT FORÉZIEN
OPINEL
MIROR
ALCALI
S.D.C
Cadum
CHARBON DE BOIS
ENGRAIS DE FOND
Quincaillerie en kit 66.50€
(Vitrine montée
Tous les objets sont à fabriquer

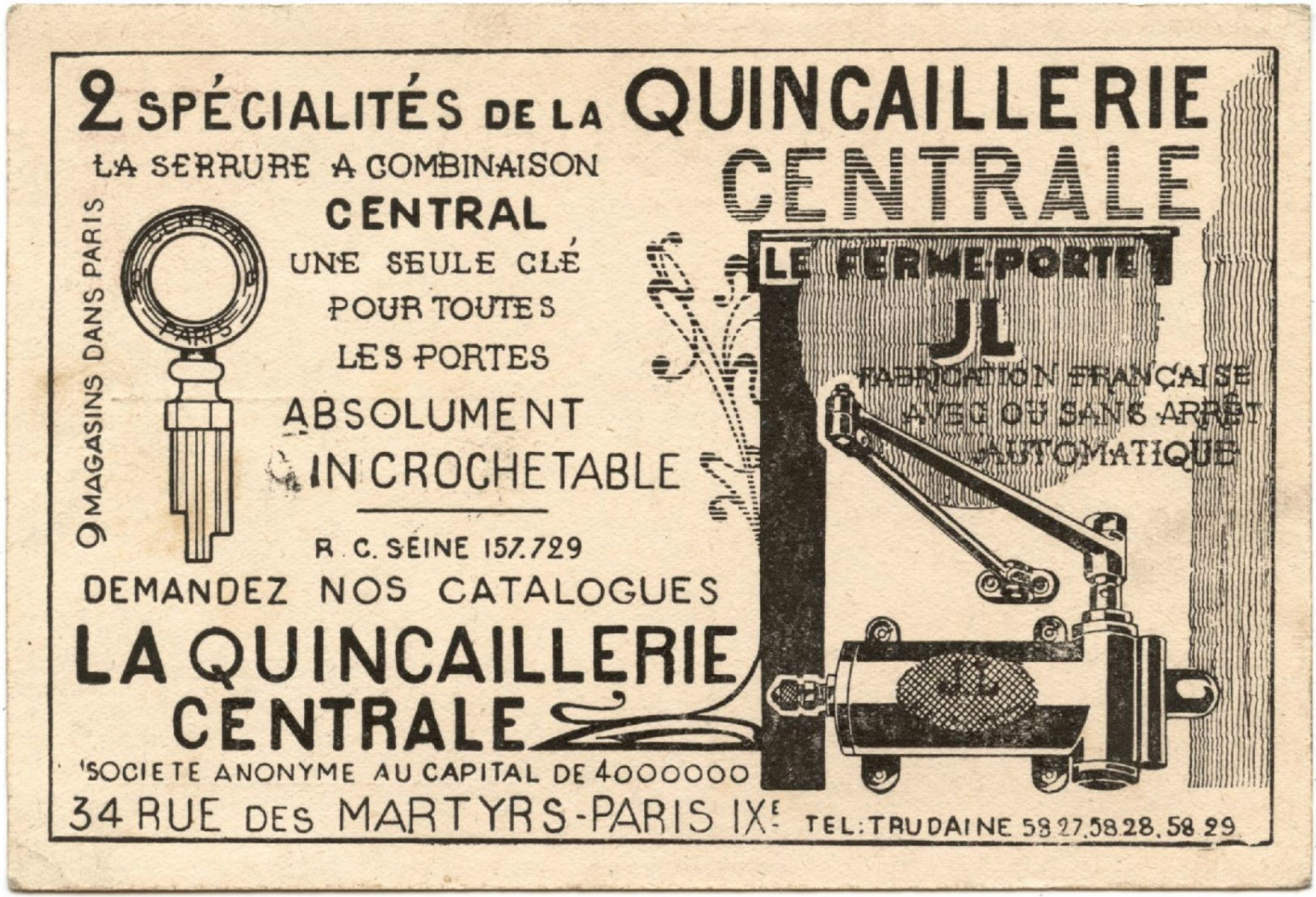
COURS 3

A) Namespace et opérateur ::
B) Flots console, affichages cout<<
C) Flots console, saisies cin>>
D) Type bool, littéral nullptr
E) Les chaînes std::string
F) Les conteneurs std::vector< >
G) Déclaration variables et portée
H) Alias de type, déclaration auto

COURS 3

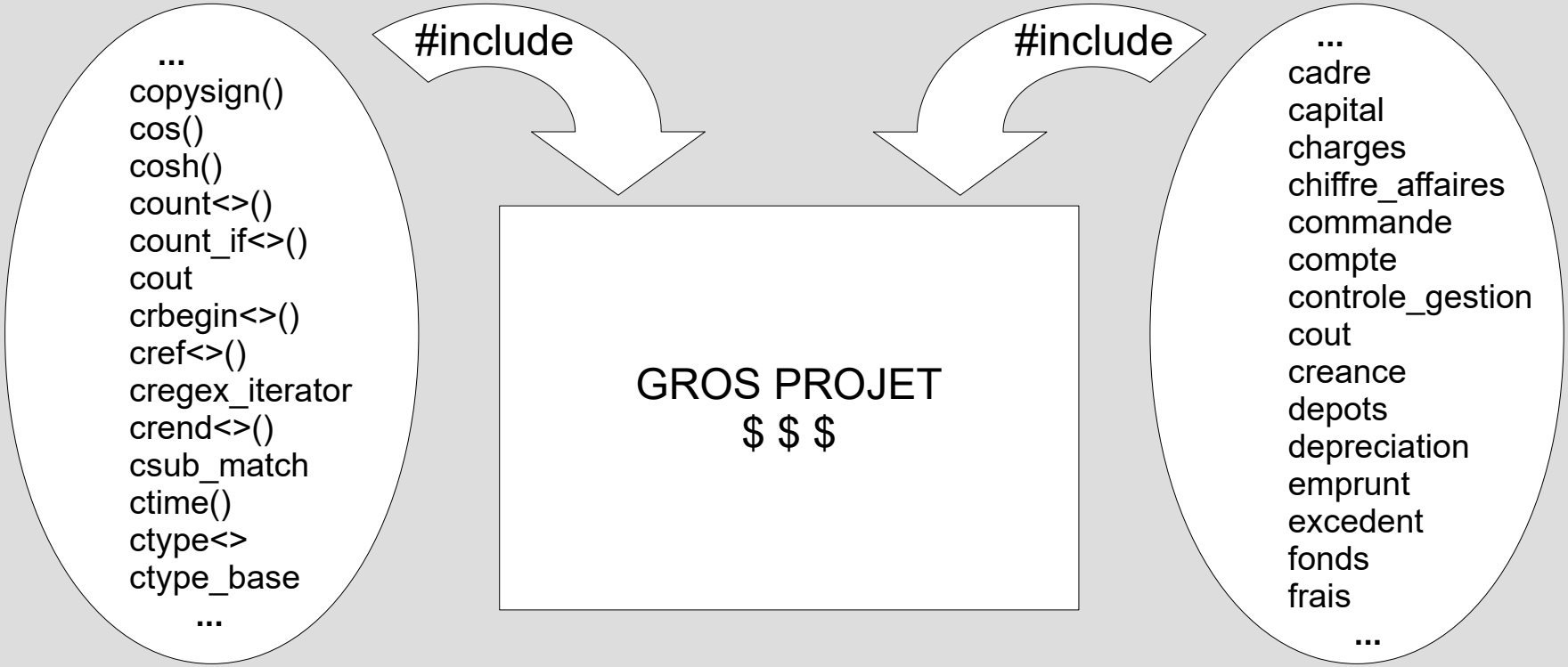
A) Namespace et opérateur ::
B) Flots console, affichages cout<<
C) Flots console, saisies cin>>
D) Type bool, littéral nullptr
E) Les chaînes std::string
F) Les conteneurs std::vector< >
G) Déclaration variables et portée
H) Alias de type, déclaration auto

Namespace et opérateur ::



Namespace et opérateur ::

- *L'expérience en développement de gros projets montre que les « collisions d'identifiants » sont fréquentes quand on réunit ensemble de grosses bases de code*

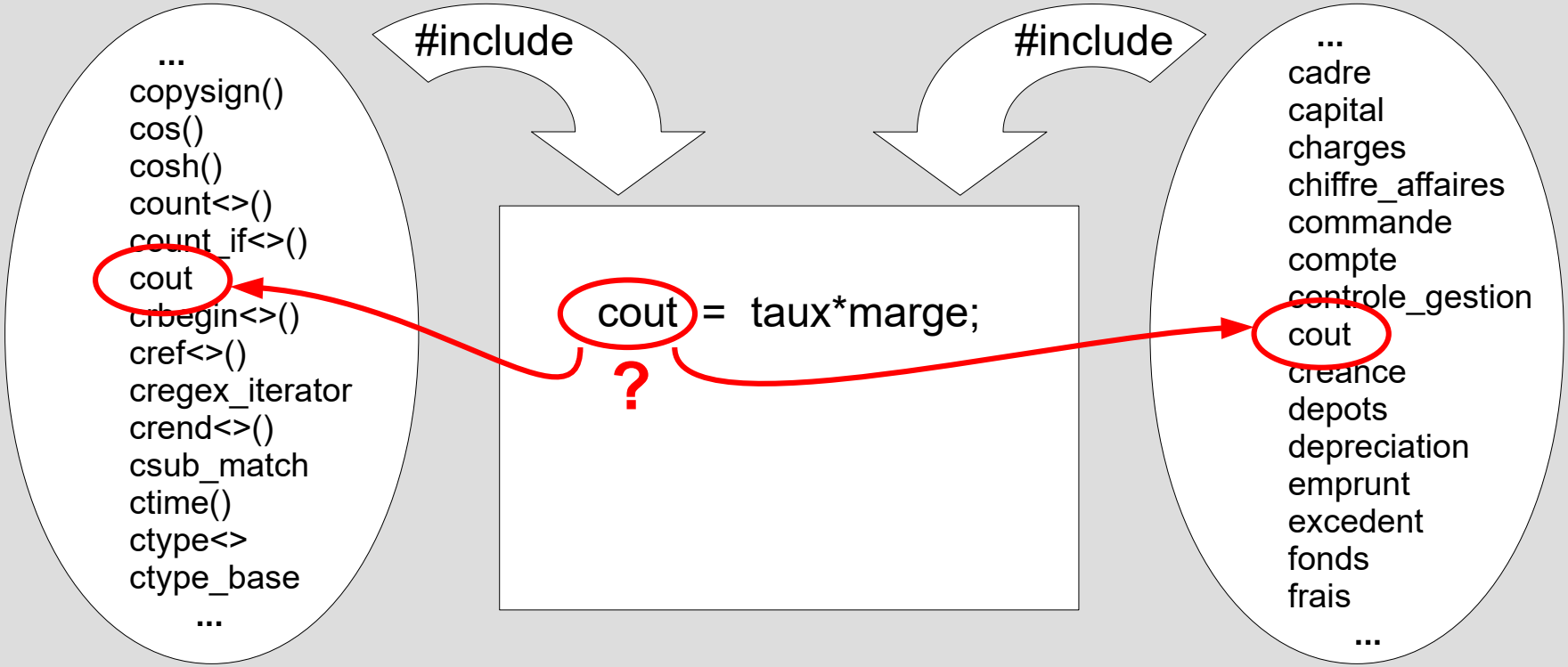


Librairie standard du C++
~ 1000 identifiants

Code métier « compta »
~ 1000 identifiants

Namespace et opérateur ::

- *L'expérience en développement de gros projets montre que les « collisions d'identifiants » sont fréquentes quand on réunit ensemble de grosses bases de code*



Librairie standard du C++
~ 1000 identifiants

Code métier « compta »
~ 1000 identifiants

Namespace et opérateur ::

- *Par exemple la bibliothèque standard du C++ définit un identifiant* **`cout`** *(Character OUTput) tandis qu'on peut facilement imaginer qu'un vieux (mais précieux) code de gestion de comptabilité écrit en C en 1989 définit une variable ou une constante de coût (combien ça coûte) nommée aussi* **`cout`**
- *Lorsqu'on va vouloir mettre dans un même projet à la fois la bibliothèque standard du C++ et ce code C le* ***compilateur*** *va dans le meilleur des cas faire une erreur (compréhensible ou pas)...*
  - *error: conflicting declaration 'float cout'*
  - *error: invalid operands of types 'float' and 'const char [13]' to binary 'operator<<'*

Namespace et opérateur ::

- *Ou pire, alors que le développeur pensait à l'un des identifiant le compilateur va réussir à compiler le code sur la base de l'autre identifiant que souvent le développeur ne connaît même pas ! Et c'est le chaos...*
- *Le problème devient encore plus fréquent quand on a des versions différentes de bibliothèques et qu'on veut mettre ensemble un code qui utilise une vieille version avec un code qui utilise une version récente (et le manager n'est pas d'accord pour tout recoder)*
- *Sur le terrain du développement de grosses applications ces situations sont assez courantes et coûtent assez cher pour qu'on s'astreigne à la fastidieuse* ***discipline des namespace...***

Namespace et opérateur ::

- *Un namespace est un espace de nommage !*
- *Le namespace lui même a un identifiant, les sections de code concernées par ce namespace sont encadrées par une déclaration de namespace*
- *A l'intérieur d'un bloc namespace on accède normalement aux identifiants de ce namespace*

factures.h

```
namespace compta
{
    const float tva = 19.6;
    float prixTtc(float prixHt);
}
```

factures.cpp

```
#include <factures.h>

namespace compta
{
    float prixTtc(float prixHt)
    {
        return tva * prixHt;
    }
}
```

ok

Namespace et opérateur ::

- *A l'extérieur du bloc namespace on ne voit pas les identifiants du namespace (c'est le but !)*

factures.h

```
namespace compta
{
    const float tva = 19.6;
    float prixTtc(float prixHt);
}
```

error: … was not declared in this scope

main.cpp

```
#include <factures.h>


int main()
{
    float aPayer = prixTtc(150.20);

    float doubleTva = tva * 2.0;
```

Namespace et opérateur ::

- *A l'extérieur du bloc namespace pour accéder aux identifiants du namespace il faut bien préciser de quoi on parle en préfixant par `compta::`*

factures.h

```
namespace compta
{
    const float tva = 19.6;
    float prixTtc(float prixHt);
}
```

main.cpp

```
#include <factures.h>


int main()
{
    float aPayer = compta::prixTtc(150.20);

    float doubleTva = compta::tva * 2.0;
```

ok

Namespace et opérateur ::

- *On retrouve ici un principe de la programmation OO : l'encapsulation. Un namespace est une capsule de nommage, pour y accéder on respecte un protocole.*

factures.h

```
namespace compta
{
    const float tva = 19.6;
    float prixTtc(float prixHt);
}
```

main.cpp

```
#include <factures.h>


int main()
{
    float aPayer = compta::prixTtc(150.20);

    float doubleTva = compta::tva * 2.0;
```

Namespace et opérateur ::

- *:: est l'opérateur de résolution de portée*
  scope resolution operator
  (utilisé aussi dans d'autres contextes, classes, énumérations...)

factures.h

```
namespace compta
{
    const float tva = 19.6;
    float prixTtc(float prixHt);
}
```

main.cpp

```
#include <factures.h>


int main()
{
    float aPayer = compta::prixTtc(150.20);

    float doubleTva = compta::tva * 2.0;
```

Namespace et opérateur ::

- *Si on est dans une section de code appelant qui utilise intensivement les identifiants d'un namespace il peut devenir fastidieux de taper à chaque fois `compta::`*

factures.h

```
namespace compta
{
    const float tva = 19.6;
    float prixTtc(float prixHt);
}
```

main.cpp

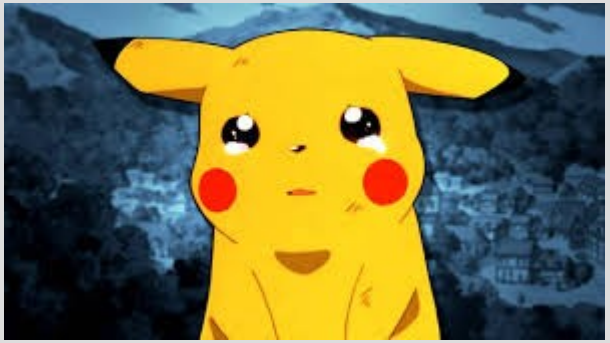
```
#include <factures.h>


int main()
{
    float aPayer = compta::prixTtc(150.20);

    float doubleTva = compta::tva * 2.0;
```

Namespace et opérateur ::

- *On peut utiliser une directive `using namespace` sous la directive les identifiants deviennent visibles !*
- *Mais ceci annule les bénéfices des namespaces...*

factures.h

```
namespace compta
{
    const float tva = 19.6;
    float prixTtc(float prixHt);
}
```



main.cpp

```
#include <factures.h>

using namespace compta;

int main()
{
    float aPayer = prixTtc(150.20);

    float doubleTva = tva * 2.0;
```

Namespace et opérateur ::

- *L'ensemble des identifiants des bibliothèques standards du C++ sont tous dans l'espace de nommage* ***`std`***
- *On écrira donc très souvent* `std::` *et il sera tentant d'utiliser une directive* ***`using namespace std;`***
- *Ainsi std::cout deviendra cout etc...*
- *Cette pratique est* ***courante*** *et acceptable sur les forums techniques pour de courts extraits de code, ou lors d'une explication sur un tableau pour des raisons de lisibilité et de compacité du code...*
- *Elle est rigoureusement interdite dans les fichiers .h car elle se propage alors aux .cpp utilisateurs du .h*

COURS 3

A) Namespace et opérateur ::
B) Flots console, affichages cout<<
C) Flots console, saisies cin>>
D) Type bool, littéral nullptr
E) Les chaînes std::string
F) Les conteneurs std::vector< >
G) Déclaration variables et portée
H) Alias de type, déclaration auto

Flots console, affichages cout<<

Flots console, affichages cout<<

- **`std::cout`** *est une instance de la classe* `std::ostream`
- *Ce qu'on lui envoie avec l'opérateur d'insertion* **<<** *s'**affiche** à la console. On peut chaîner les insertions.*

C++

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello modern world !" << std::endl;
```

C

```c
#include <stdio.h>

int main()
{
    printf("Good bye old world...\n");
```

Flots console, affichages cout<<

- *std::cout est une instance de la classe std::ostream*
- *Ce qu'on lui envoie avec l'opérateur d'insertion << apparaît à la console. On peut chaîner les insertions.*

C++

```
#include <iostream>

int main()
{
    std::cout << "Hello modern world !" << std::endl;
```

Insertion dans la console

End of Line, idem "\n"

C

```
#include <stdio.h>

int main()
{
    printf("Good bye old world...\n");
```

Flots console, affichages cout<<

- *Les contenus et valeurs à afficher se codent là où ils apparaissent dans la chaîne d'insertions*
- *Les types sont déduits, pas besoin de les préciser*

C++

```
int i = 5;
float x = 0.123456;

std::cout << "Votre entier : " << i << " Votre reel : " << x << std::endl;
```

C

```
int i = 5;
float x = 0.123456;

printf("Votre entier : %d Votre reel : %f\n", i, x);
```

Flots console, affichages cout<<

- *Les contenus et valeurs à afficher se codent là où ils apparaissent dans la chaîne d'insertions*
- *Les types sont déduits, pas besoin de les préciser*

C++

```
int i = 5;
float x = 0.123456;

std::cout << "Votre entier : " << i << " Votre reel : " << x << std::endl;
```

Valeurs indiquées localement

C

```
int i = 5;
float x = 0.123456;

printf("Votre entier : %d Votre reel : %f\n", i, x);
```

Valeurs rapportées aux formats %

Flots console, affichages cout<<

- *L'opérateur << indique le sens de circulation de l'information : messages et valeurs vers l'affichage*
- *Mais l'opérateur << s'évalue de gauche à droite !*

C++

```
int i = 5;
float x = 0.123456;

std::cout << "Votre entier : " << i << " Votre reel : " << x << std::endl;
```

Est équivalent à la séquence :

C++

```
std::cout << "Votre entier : ";
std::cout << i;
std::cout << " Votre reel : ";
std::cout << x;
std::cout << std::endl;
```

Flots console, affichages cout<<

- *On peut découper en plusieurs lignes de code pour plus de lisibilité (- de 70 caracs par ligne conseillé)*
- *Mettre le chaînage << à la ligne et non en fin de ligne*

C++

```
int i = 5;
float x = 0.123456;

std::cout << "Votre entier : " << i
          << " Votre reel : " << x
          << std::endl;
```

Flots console, affichages cout<<

- *Les formatages sont des modification de l'état de l'objet std::cout …*

```
int nbArticles = 3; float prixArticle = 2.5;
char nomArticle[] = "crochet";
```

C++

```
std::cout << std::setprecision(2)
          << std::fixed
          << "Votre commande : "
          << nbArticles << " "
          << nomArticle << "s "
          << "pour un total de "
          << nbArticles*prixArticle
          << std::endl;
```

C

```
printf("Votre commande : %d %ss pour un total de %.02f\n",
       nbArticles, nomArticle, nbArticles*prixArticle);
```

console

```
Votre commande : 3 crochets pour un total de 7.50
```

Flots console, affichages cout<<

- *Les formatages sont des modification de l'état de l'objet std::cout …*

```
int nbArticles = 3; float prixArticle = 2.5;
char nomArticle[] = "crochet";
```

C++

```
std::cout << std::setprecision(2)
          << std::fixed
          << "Votre commande : "
          << nbArticles << " "
          << nomArticle << "s "
          << "pour un total de "
          << nbArticles*prixArticle
          << std::endl;
```

Objet « flot de sortie » modifié

Formatage ponctuel

C

```
printf("Votre commande : %d %ss pour un total de %.02f\n",
       nbArticles, nomArticle, nbArticles*prixArticle);
```

console

```
Votre commande : 3 crochets pour un total de 7.50
```

Flots console, affichages cout<<

- *Les formatages sont des modification de l'état de l'objet std::cout …*

```
int nbArticles = 3; float prixArticle = 2.5;
char nomArticle[] = "crochet";
```

C++

```
std::cout << std::setprecision(2)
          << std::fixed
          << "Votre commande : "
          << nbArticles << " "
          << nomArticle << "s "
          << "pour un total de "
          << nbArticles*prixArticle
          << std::endl;
...
...
...

float tab[3] = {2, 5.1, 2.49};

std::cout << tab[0] << " " << tab[1] << " " << tab[2] << std::endl;
```

Objet « flot de sortie » modifié

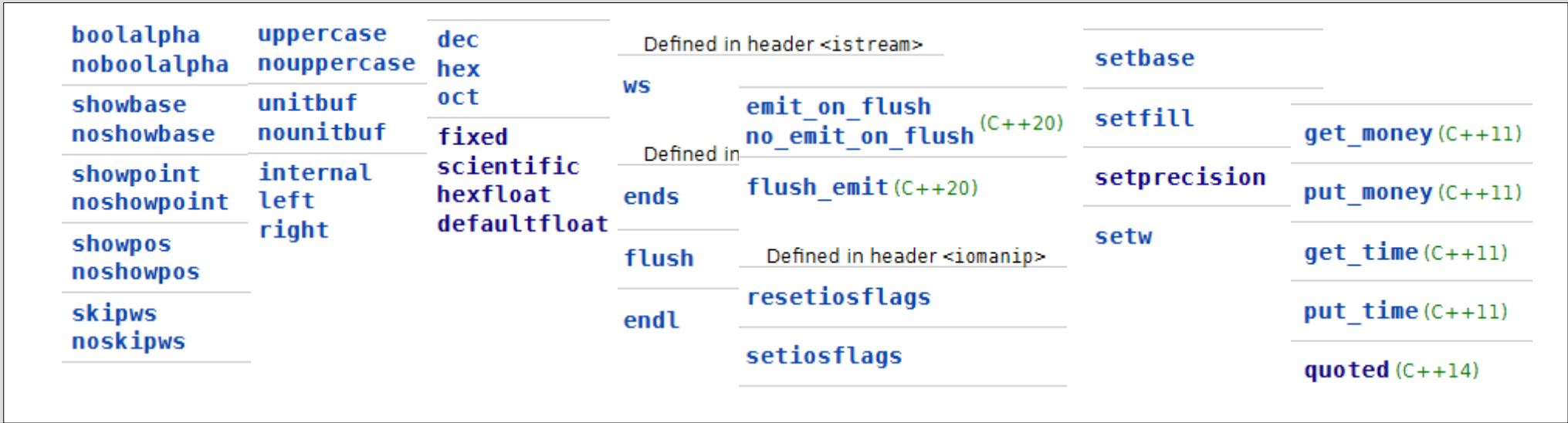
Le résultat dépend de la présence ou non du code au dessus...

Flots console, affichages cout<<

- *Rassurez vous on ne va pas vous taquiner en DS sur les subtilités des formatages avec* `cout<<`

  Pour les TPs il faut retenir que :

- *Certains formats nécessitent* `#include <iomanip>`
- *Le catalogue n'est pas à connaître par cœur mais vous devez savoir consulter une doc technique : https://en.cppreference.com/w/cpp/io/manip*



COURS 3

A) Namespace et opérateur ::
B) Flots console, affichages cout<<
C) Flots console, saisies cin>>
D) Type bool, littéral nullptr
E) Les chaînes std::string
F) Les conteneurs std::vector< >
G) Déclaration variables et portée
H) Alias de type, déclaration auto

Flots console, saisies cin>>

Flots console, saisies cin>>

- **`std::cin`** *est une instance de la classe* `std::istream`
- *Les données qu'il envoie vers des variables avec l'opérateur d'extraction* **>>** *correspondent aux* ***saisies***

C++

```
int i;
float x;

std::cout << "Veuillez saisir un entier puis un reel" << std::endl;
std::cin >> i >> x;

std::cout << "Votre entier vaut " << i
          << " et votre reel " << x << std::endl;
```

C

```
int i;
float x;

printf("Veuillez saisir un entier puis un reel\n");
scanf("%d", &i);
scanf("%f", &x);

printf("Votre entier vaut %d et votre reel %f\n", i, x);
```

Flots console, saisies cin>>

- *Evidemment ce code plutôt lisible échoue lamentablement si l'utilisateur entre autre chose que ce qui est attendu...*

Ce code compile sans error ni warning...

C++

```
int i;
float x;

std::cout << "Veuillez saisir un entier puis un reel" << std::endl;
std::cin >> i >> x;

std::cout << "Votre entier vaut " << i
          << " et votre reel " << x << std::endl;
```

console

```
Veuillez saisir un entier puis un reel
Surprise !
Votre entier vaut 0 et votre reel 3.7651e-039

Process returned 0 (0x0)   execution time : 5.413 s
Press any key to continue.
```

x non initialisé après cin !

Flots console, saisies cin>>

- *La programmation de saisies utilisateur « blindées » en mode console reste toujours une punition pour le développeur parce que :*
  - *L'utilisateur entre ce qu'il veut (on ne peut pas physiquement bloquer les touches. Avec les écrans tactiles c'est différent...)*
  - *Le langage ne peut pas décider à la place du programmeur ce que l'application doit faire quand l'utilisateur n'est pas dans les clous.*
  - ***Basiquement il faudrait saisir ligne par ligne dans des chaînes puis analyser ces chaînes reçues caractère par caractère (parsing).***
  - *Ceci fera l'objet d'un exercice au TD/TP 3...*

Flots console, saisies cin>>

- *Si vous bloquez avec une fonctionnalité C++ que vous sauriez faire en C ...*
- *Si une entité d'une bibliothèque C n'a pas d'équivalent ou est plus adaptée que l'équivalent C++ …*
- *Il est toujours possible d'utiliser du C dans du C++ !*
- *Dans ce cas faites un include du nom de la bibliothèque C sans le .h et préfixée par c*

`stdio.h → cstdio` `math.h → cmath` `...`

C++

N'en abusez pas
On est là pour le C++ !

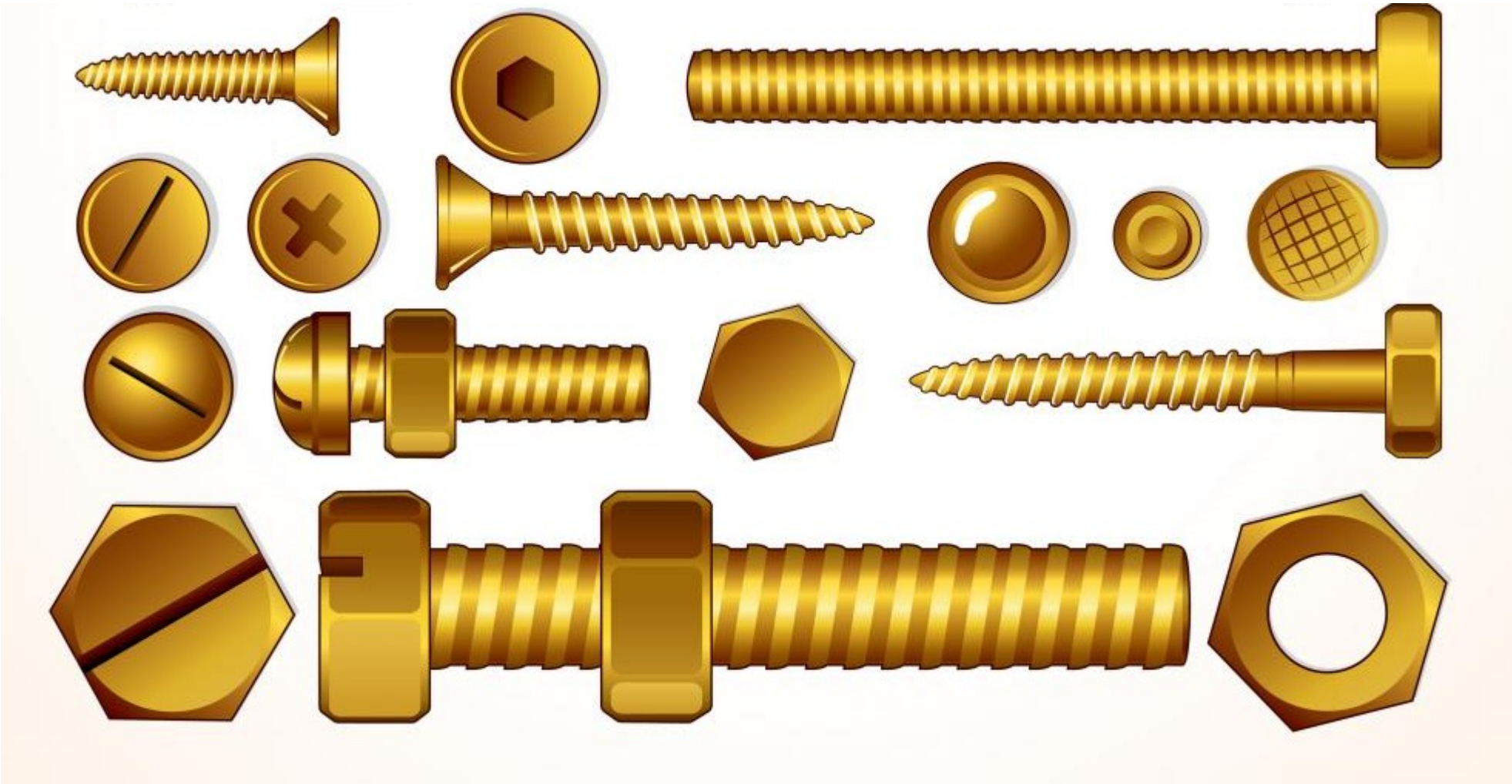
```
#include <cstdio>

int main()
{

    float a = 1.2345;

    printf("%.01f %.02f %.03f\n", a, a, a);
```

COURS 3

A) Namespace et opérateur ::
B) Flots console, affichages cout<<
C) Flots console, saisies cin>>
D) Type bool, littéral nullptr
E) Les chaînes std::string
F) Les conteneurs std::vector< >
G) Déclaration variables et portée
H) Alias de type, déclaration auto

Type bool, littéral nullptr

- *Le C++ introduit de nouveaux types et de nouveaux mots clés pour des valeurs élémentaires*



Type bool, littéral nullptr

- *Le type* **`bool`** *prend les valeurs* **`true`** *ou* **`false`**
- *C++ peut convertir implicitement bool depuis-vers les types numériques / pointeurs (=0 ↔ false ≠0 ↔ true)*

C++

```
bool estVert = true;
bool seDeplaceSousLeau = false;

if ( estVert && seDeplaceSousLeau )
    std::cout << "Un choux-marin !" << std::endl;
else
    std::cout << "Je ne sais pas..." << std::endl;
```

C

```
char estVert = 1;
char seDeplaceSousLeau = 0;

if ( estVert && seDeplaceSousLeau )
    printf("Un choux-marin !\n");
else
    printf("Je ne sais pas...\n");
```

Type bool, littéral nullptr

- *La valeur spéciale* **nullptr** *remplace* **NULL** *pour indiquer un pointeur qui pointe sur « rien »*
- **nullptr** *a un type pointeur* **NULL** *un type entier...*

C++

```
struct Coords
{
    double x, y;
};

int main()
{
    Coords* ici = nullptr;

    // ...
    // ici est alloué ou pas...
    // ...

    if ( ici != nullptr )
        /// Ok, ici pointe quelque chose
        std::cout << ici->x << " " << ici->y << std::endl;
```

COURS 3

A) Namespace et opérateur ::
B) Flots console, affichages cout<<
C) Flots console, saisies cin>>
D) Type bool, littéral nullptr
E) Les chaînes std::string
F) Les conteneurs std::vector< >
G) Déclaration variables et portée
H) Alias de type, déclaration auto

Les chaînes std::string

Les chaînes std::string

- *Le type* **`std::string`** *de la bibliothèque* **`string`**
- *Quelle taille fait le tableau de* `char` *qui hébergera une chaîne ? La taille qu'il faut !*

C++

```
Nom et Prenom SVP
Pig George
Bonjour George Pig
Ton prenom a 6 lettres
```

```cpp
#include <iostream>
#include <string>

int main()
{

    std::string nom, prenom;

    std::cout << "Nom et Prenom SVP" << std::endl;

    std::cin >> nom;
    std::cin >> prenom;

    std::cout << "Bonjour " << prenom << " " << nom << std::endl;

    std::cout << "Ton prenom a " << prenom.size() << " lettres"
              << std::endl;
```

Les chaînes std::string

- *La machinerie complexe de l'objet commence à payer !*
- *Chaque chaîne est un objet de la classe* **`std::string`** *avec des attributs privés et des méthodes publiques*

C++

```
Nom et Prenom SVP
Pig George
Bonjour George Pig
Ton prenom a 6 lettres
```

```cpp
#include <iostream>
#include <string>

int main()
{

    std::string nom, prenom;

    std::cout << "Nom et Prenom SVP" << std::endl;

    std::cin >> nom;
    std::cin >> prenom;

    std::cout << "Bonjour " << prenom << " " << nom << std::endl;

    std::cout << "Ton prenom a " << prenom.size() << " lettres"
              << std::endl;
```

Les chaînes std::string

- *Un* **`std::string`** *est un « objet valeur »*
- *Le type se comporte* ***comme un type scalaire****, les affectations / comparaisons se font par valeur*

C++

```
Deux Un
```

```
std::string x;
std::string y;

x = "Un";
y = x;
x = "Deux";

std::cout << x << " " << y << std::endl;
```

Les chaînes std::string

- *Un* **`std::string`** *est un « objet valeur »*
- *Le type se comporte* ***comme un type scalaire****, les affectations / comparaisons se font par valeur*

C++

```
x et y ont meme valeur
x a la valeur "ABC"
x et y n'ont pas meme adresse
```

```cpp
std::string x;
std::string y;

x = "ABC";
y = "ABC";

if ( x == y )
    std::cout << "x et y ont meme valeur" << std::endl;

if ( x == "ABC" )
    std::cout << "x a la valeur \"ABC\"" << std::endl;

if ( &x != &y )
    std::cout << "x et y n'ont pas meme adresse" << std::endl;
```

Les chaînes std::string

- *Des méthodes sous forme explicite ou sous forme d'opérateurs permettent de manipuler les* **`std::string`**
- *On se **moque** des problèmes d'allocation...*

C++

```
This is Big !
This is Bigger !
I have Big hunger !
For a Burger !
```

```cpp
std::string nom = "Pig";

nom.front() = 'B';
std::cout << "This is " << nom << " !" << std::endl;

nom += "ger";
std::cout << "This is " << nom << " !" << std::endl;

nom.insert(3, " hun");
std::cout << "I have " << nom << " !" << std::endl;

nom.erase(1, 4);
nom[2] = 'r';
std::cout << "For a " << nom << " !" << std::endl;
```

Les chaînes std::string

- *Des méthodes sous forme explicite ou sous forme d'opérateurs permettent de manipuler les* **`std::string`**
- *On se **moque** des problèmes d'allocation...*

C++

```
std::string nom = "Pig";
```
"Pig"

```
nom.front() = 'B';
```
"Pig" → "Big"

```
nom += "ger";
```
"Big" → "Bigger"

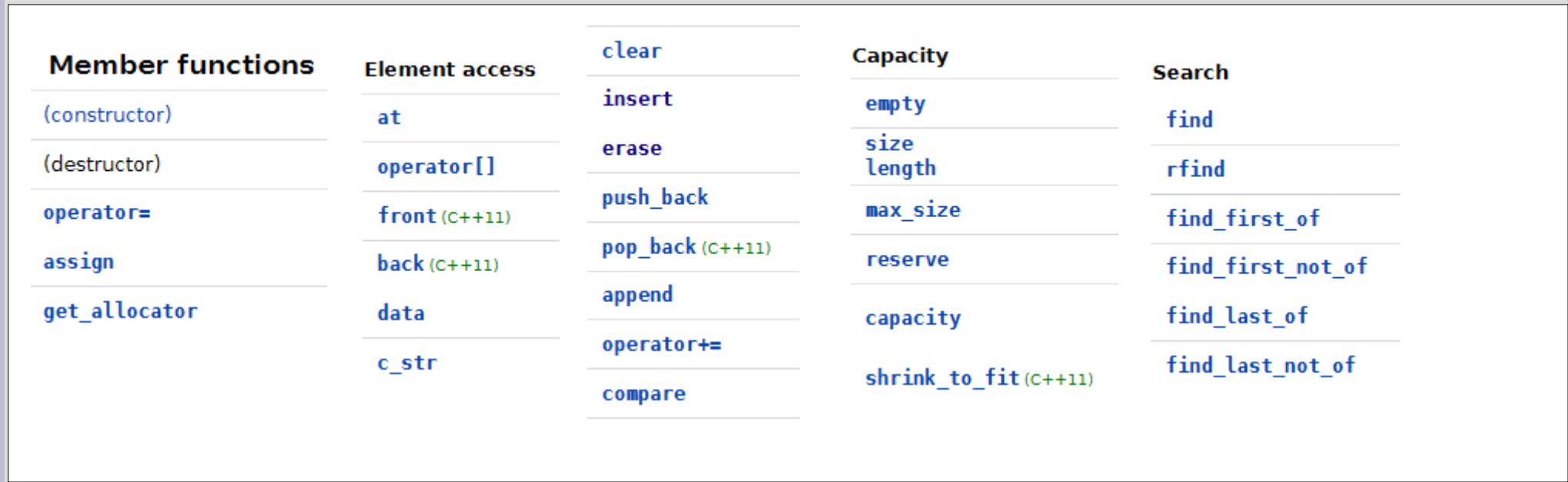
```
nom.insert(3, " hun");
```
"Bigger" → "Big hunger"

```
nom.erase(1, 4);
```
"Big hunger" → "Bunger"

```
nom[2] = 'r';
```
"Bunger" → "Burger"

Les chaînes std::string

- *Des méthodes sous forme explicite ou sous forme d'opérateurs permettent de manipuler les* **`std::string`**
- *Le catalogue n'est pas à connaître par cœur mais vous devez savoir consulter une doc technique : https://en.cppreference.com/w/cpp/string/basic_string*



Member functions

(constructor)
(destructor)
operator=
assign
get_allocator

Element access

at
operator[]
front (C++11)
back (C++11)
data
c_str

clear
insert
erase
push_back
pop_back (C++11)
append
operator+=
compare

Capacity

empty
size
length
max_size
reserve
capacity
shrink_to_fit (C++11)

Search

find
rfind
find_first_of
find_first_not_of
find_last_of
find_last_not_of

Les chaînes std::string

- *Les prototypes des méthodes et fonctions C++ de la bibliothèque standard sont notoirement illisibles...*
- *Ici déclaration de* **`std::getline`** *qui lit une ligne complète dans une string (cin>> coupe aux espaces)*

std::getline

Defined in header <string>

```
template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>& input,
                                           std::basic_string<CharT,Traits,Allocator>& str,
                                           CharT delim );

template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>&& input,
                                           std::basic_string<CharT,Traits,Allocator>& str,
                                           CharT delim );

template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>& input,
                                           std::basic_string<CharT,Traits,Allocator>& str );

template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>&& input,
                                           std::basic_string<CharT,Traits,Allocator>& str );
```

getline reads characters from an input stream and places them into a string:

Les chaînes std::string

- *Concentrez vous sur les **exemples***
  https://fr.cppreference.com/w/cpp/string/basic_string/getline
- *Ici utilisation de* **`std::getline`** *qui lit une ligne complète dans une string (cin>> coupe aux espaces)*

Exemple

Le code suivant demande à l'utilisateur son nom, puis les salue l'aide de ce nom .

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

int main()
{
    std::string name;
    std::cout << "What is your name? ";
    std::getline(std::cin, name);
    std::cout << "Hello " << name << ", nice to meet you.";
}
```

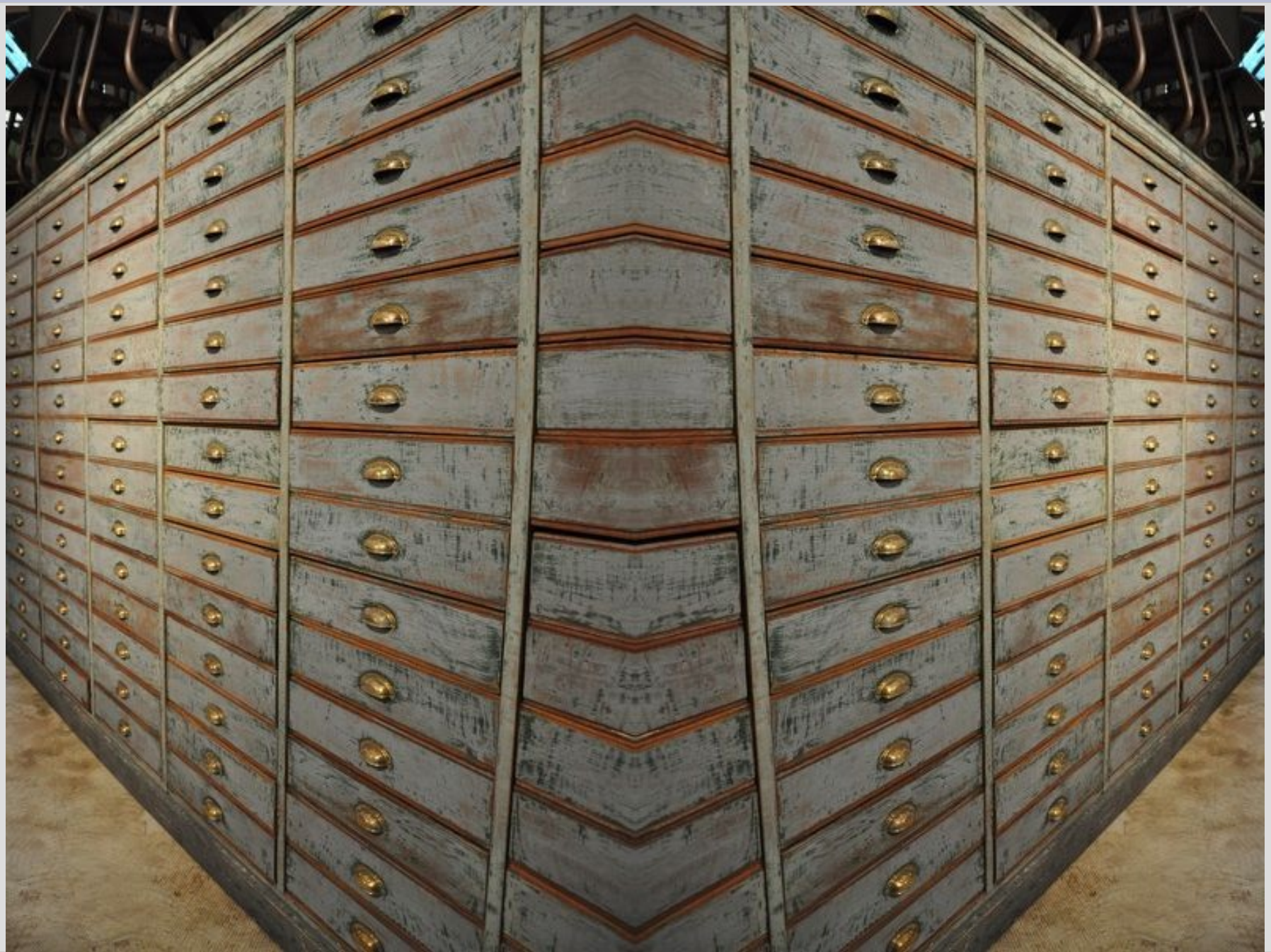
Résultat possible :

```
What is your name? John Q. Public
Hello John Q. Public, nice to meet you.
```

COURS 3

A) Namespace et opérateur ::
B) Flots console, affichages cout<<
C) Flots console, saisies cin>>
D) Type bool, littéral nullptr
E) Les chaînes std::string
F) Les conteneurs std::vector< >
G) Déclaration variables et portée
H) Alias de type, déclaration auto

Les conteneurs std::vector< >



Les conteneurs std::vector< >

- *C++ offre la possibilité d'utiliser des types paramétrés en types (templates et programmation générique)*
- *Un des plus utiles est* **`std::vector< … >`**

C++

```
0 elements
3 elements
1.23 / 2.5 / 3.15
```

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<float> monVec;

    std::cout << monVec.size() << " elements" << std::endl;

    monVec.push_back(1.23);
    monVec.push_back(2.5);
    monVec.push_back(3.15);

    std::cout << monVec.size() << " elements" << std::endl;
    std::cout << monVec[0] << " / "
              << monVec[1] << " / "
              << monVec[2] << std::endl;
```

Les conteneurs std::vector< >

- *Le vecteur est un type **conteneur**, il y en a d'autres, on fera un cours complet à ce sujet !*
- *Comme un tableau … extensible !*

C++

```
0 elements
3 elements
1.23 / 2.5 / 3.15
```

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<float> monVec;

    std::cout << monVec.size() << " elements" << std::endl;

    monVec.push_back(1.23);
    monVec.push_back(2.5);
    monVec.push_back(3.15);

    std::cout << monVec.size() << " elements" << std::endl;
    std::cout << monVec[0] << " / "
              << monVec[1] << " / "
              << monVec[2] << std::endl;
```

Les conteneurs std::vector< >

- *Des méthodes sous forme explicite ou sous forme d'opérateurs permettent de manipuler les* **`vector`**
- *Le catalogue n'est pas à connaître par cœur mais vous devez savoir consulter une doc technique : https://en.cppreference.com/w/cpp/container/vector*

Member functions

(constructor)	constructs the `vector` (public member function)
(destructor)	destructs the `vector` (public member function)
`operator=`	assigns values to the container (public member function)
`assign`	assigns values to the container (public member function)
`get_allocator`	returns the associated allocator (public member function)

Element access

`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data` (C++11)	direct access to the underlying array (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)
`reserve`	reserves storage (public member function)
`capacity`	returns the number of elements that can be held in ((public member function)
`shrink_to_fit` (C++11)	reduces memory usage by freeing unused memory (public member function)

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements (public member function)
`emplace` (C++11)	constructs element in-place (public member function)
`erase`	erases elements (public member function)
`push_back`	adds an element to the end (public member function)
`emplace_back` (C++11)	constructs an element in-place at the end (public member function)
`pop_back`	removes the last element (public member function)
`resize`	changes the number of elements stored (public member function)
`swap`	swaps the contents (public member function)

Non-member functions

`operator==` `operator!=` `operator<` `operator<=` `operator>` `operator>=`	lexicographically compares the values in the vector (function template)

Les conteneurs std::vector< >

- *Les méthodes* **`size()`** *des objets* **`string`** *et* **`vector`** *sont de type* **`size_t`** *(je simplifie, chaque classe à son size_type...)*
- *Utiliser ce type pour les compteurs comparés à* **`size()`**

C++

```
0, 1, 1, 2, 3, 5, 8, 13, ...
```

```cpp
size_t n = 8;
size_t i;


std::vector<int> fibo;
fibo.push_back(0);
fibo.push_back(1);

for (i=2; i<n; ++i)
    fibo.push_back( fibo[i-2] + fibo[i-1] );

for (i=0; i<fibo.size(); ++i)
    std::cout << fibo[i] << ", ";

std::cout << "..." << std::endl;
```

Les conteneurs std::vector< >

- *Les méthodes* **`size()`** *des objets* **`string`** *et* **`vector`** *sont de type* **`size_t`** *(je simplifie, chaque classe à son size_type...)*
- *Utiliser ce type pour les compteurs comparés à* **`size()`**

C++

```
0, 1, 1, 2, 3, 5, 8, 13, ...
```

```cpp
size_t n = 8;
size_t i;


std::vector<int> fibo;
fibo.push_back(0);
fibo.push_back(1);

for (i=2; i<n; ++i)
    fibo.push_back( fibo[i-2] + fibo[i-1] );

for (i=0; i<fibo.size(); ++i)
    std::cout << fibo[i] << ", ";

std::cout << "..." << std::endl;
```

Effet de mode (sans importance)
en C++ on préfère ++i à i++

Les conteneurs std::vector< >

- *On peut toujours utiliser un tableau classique quand on connaît le nombre d'éléments « en dur »*
- *Un tableau est toujours passé « par référence »*

C++

```
void testerTableau(int param[3])
{
    std::cout << "sous prog. tableau avant " << param[0] << std::endl;
    ++param[0];
    std::cout << "sous prog. tableau apres " << param[0] << std::endl;
}

int main()
{
    int tab[3] = {10, 20, 30};

    std::cout << "appelant tableau avant " << tab[0] << std::endl;
    testerTableau(tab);
    std::cout << "appelant tableau apres " << tab[0] << std::endl;
```

```
appelant tableau avant 10
sous prog. tableau avant 10
sous prog. tableau apres 11
appelant tableau apres 11
```

L'appelé **peut modifier**
des données de l'appelant

Les conteneurs std::vector< >

- *On peut aussi utiliser un vecteur, les mêmes syntaxes d'**initialisation à la déclaration** sont utilisables*
- *Par défaut un vecteur est passé « par valeur » : **copie***

C++

```
void testerVecteur(std::vector<int> param)
{
    std::cout << "sous prog. vecteur avant " << param[0] << std::endl;
    ++param[0];
    std::cout << "sous prog. vecteur apres " << param[0] << std::endl;
}

int main()
{
    std::vector<int> vec = {10, 20, 30};

    std::cout << "appelant vecteur avant " << vec[0] << std::endl;
    testerVecteur(vec);
    std::cout << "appelant vecteur apres " << vec[0] << std::endl;
```

Copie du vecteur de l'appelant

```
appelant vecteur avant 10
sous prog. vecteur avant 10
sous prog. vecteur apres 11
appelant vecteur apres 10
```

L'appelé n'a **pas modifié** des données de l'appelant

Les conteneurs std::vector< >

- *Comme pour* `std::string` , `std::vector` *a une sémantique par valeur, affectation et appel copient*
- *On peut passer « par référence » : voir cours suivant...*

Référence vecteur de l'appelant

C++

```
void testerVecteur(std::vector<int>& param)
{
    std::cout << "sous prog. vecteur avant " << param[0] << std::endl;
    ++param[0];
    std::cout << "sous prog. vecteur apres " << param[0] << std::endl;
}

int main()
{
    std::vector<int> vec = {10, 20, 30};

    std::cout << "appelant vecteur avant " << vec[0] << std::endl;
    testerVecteur(vec);
    std::cout << "appelant vecteur apres " << vec[0] << std::endl;
```

```
appelant vecteur avant 10
sous prog. vecteur avant 10
sous prog. vecteur apres 11
appelant vecteur apres 11
```

L'appelé **peut modifier**
des données de l'appelant

Les conteneurs std::vector< >

- *Dans les chevrons < ... > on met le type contenu*
  brackets (ou angle brackets)
- *vector peut contenir n'importe quel type copiable...*

C++

```
std::vector<std::string> mots{ "Un", "vecteur" };

mots.push_back( "de chaines" ) ;

for (size_t i=0; i<mots.size(); ++i)
    std::cout << mots[i] << " ";

std::cout << std::endl;
```

```
Un vecteur de chaines
```

Les conteneurs std::vector< >

- *Dans les chevrons < ... > on met le type contenu*
  brackets (ou angle brackets)
- *vector dans vector : tableaux à 2 dimensions*

C++

```
std::vector<std::vector<int>> mat{ {1,2}, {3, 4, 5} };

mat.push_back( std::vector<int>{6, 7, 8, 9} );

for (size_t i=0; i<mat.size(); ++i)
{
    for (size_t j=0; j<mat[i].size(); ++j)
        std::cout << mat[i][j] << " ";
    std::cout << std::endl;
}
```

```
1 2
3 4 5
6 7 8 9
```

Les conteneurs std::vector< >

- *Le vector peut être dimensionné à sa déclaration*
- *C++11 offre de nombreuses syntaxes de déclaration avec quelques pièges... on reviendra dessus en TP !*

C++

```
std::vector<float> vec1;
std::vector<float> vec2(5);
std::vector<float> vec3(5, 3.14);
std::vector<float> vec4{5, 3.14};
```

```
vec1 a 0 elements :
vec2 a 5 elements : 0.00 0.00 0.00 0.00 0.00
vec3 a 5 elements : 3.14 3.14 3.14 3.14 3.14
vec4 a 2 elements : 5.00 3.14
```

COURS 3

A) Namespace et opérateur ::
B) Flots console, affichages cout<<
C) Flots console, saisies cin>>
D) Type bool, littéral nullptr
E) Les chaînes std::string
F) Les conteneurs std::vector< >
G) Déclaration variables et portée
H) Alias de type, déclaration auto

Déclaration variables et portée

Déclaration variables et portée

- *En C++ on déclare de nouvelles variables où on veut dans un **bloc { }** pas forcément « au début »*
- *C'est aussi valable en C99 (on ne vous l'a pas dit ?)*

C++

```cpp
int main()
{
    std::string nom;
    std::cout << "Nom SVP : ";
    std::cin  >> nom;

    int age;
    std::cout << "Age SVP : ";
    std::cin  >> age;

    bool fumeur;
    std::cout << "Fumeur (true/false) : ";
    std::cin  >> std::boolalpha >> fumeur;

    if ( age>40 && fumeur )
        std::cout << "Faites une radio des poumons "
                  << nom << std::endl;
```

```
Nom SVP : Joe
Age SVP : 45
Fumeur (true/false) : true
Faites une radio des poumons Joe
```

Déclaration variables et portée

- *En C++ on déclare de nouvelles variables où on veut dans un **bloc** { } pas forcément « au début »*
- *La variable est **visible** jusqu'à la fin du bloc → }*

C++

```
int main()
{
    std::string nom;
    std::cout << "Nom SVP : ";
    std::cin  >> nom;

    int age;
    std::cout << "Age SVP : ";
    std::cin  >> age;

    bool fumeur;
    std::cout << "Fumeur (true/false) : ";
    std::cin  >> std::boolalpha >> fumeur;

    if ( age>40 && fumeur )
        std::cout << "Faites une radio des poumons "
                  << nom << std::endl;
    ...
}
```

```
Nom SVP : Joe
Age SVP : 45
Fumeur (true/false) : true
Faites une radio des poumons Joe
```

int age

Déclaration variables et portée

- Attention il y a quand même des règles et des pièges
- *La variable la plus locale cache la plus globale...*
- *La portée d'une variable (**scope**)*

C++

```
std::string x = "Globale";

int main()
{
    std::cout << x << std::endl;
    float x = 3.14;
    std::cout << x << std::endl;

    for (int i=1; i<=3; i++)
    {
        int x = 5*i;
        std::cout << x << std::endl;
    }

    std::cout << x << std::endl;

    ...
}
```

std::string x

float x

int x

```
Globale
3.14
5
10
15
3.14
```

Déclaration variables et portée

- Attention il y a quand même des règles et des pièges
- *Les variables « compteur de boucle » peuvent se déclarer localement au bloc initialisation de la boucle...*

C++

```
int main()
{



    for (int i=1; i<=3; i++)
    {
        int x = 5*i;
        std::cout << x << std::endl;
    }


    ...
}
```

```
5
10
15
```

Déclaration variables et portée

- Attention il y a quand même des règles et des pièges
- *Les variables « compteur de boucle » peuvent se déclarer localement au bloc initialisation de la boucle...*

C++

```
int main()
{




    for (int i=1; i<=3; i++)
    {
        int x = 5*i;
        std::cout << x << std::endl;
    }

    std::cout << i << std::endl;

    ...
}
```

```
error: 'i' was not declared in this scope
```

Déclaration variables et portée

- Attention il y a quand même des règles et des pièges
- *Ça correspond à une logique de localité :*
  restreindre les variables à la plus petite portée utile

C++

contexte

variable i déclarée ou pas ?
de type int si déclarée ?

Ceci est une « unité d'exécution »
qui peut être dupliquée ou déplacée
sans dépendre d'un contexte

```
for (int i=1; i<=3; i++)
{
    int x = 5*i;
    std::cout << x << std::endl;
}
```

Déclaration variables et portée

- Attention il y a quand même des règles et des pièges
- *Les* `case` *dans les switch ne sont pas des blocs !*

C++

```cpp
int main()
{
    int choix;
    std::cin >> choix;

    switch (choix)
    {
        case 1:
            int x = 3;
            std::cout << x;
            break;

        case 2:
            int x = 6;
            std::cout << x;
            break;
    }
}
```

```
error: redeclaration of 'int x'
```

Déclaration variables et portée

- Attention il y a quand même des règles et des pièges
- *Les* `case` *dans les switch ne sont pas des blocs !*

C++

```cpp
int main()
{
    int choix;
    std::cin >> choix;

    switch (choix)
    {
        case 1:
            int x = 3;
            std::cout << x;
            break;

        case 2:
            int y = 6;
            std::cout << y;
            break;
    }
}
```

```
error: jump to case label
       crosses initialization of 'int x'
```

Déclaration variables et portée

- Attention il y a quand même des règles et des pièges
- *Les* `case` *dans les switch ne sont pas des blocs !*
  Si ça fait sens de déclarer localement : préciser **{ }**

C++

```
switch (choix)
{
    case 1:
    {
        int x = 3;              Ok ça compile !
        std::cout << x;
        break;
    }

    case 2:
    {
        int x = 6;
        std::cout << x;
        break;
    }
}
```

COURS 3

A) Namespace et opérateur ::
B) Flots console, affichages cout<<
C) Flots console, saisies cin>>
D) Type bool, littéral nullptr
E) Les chaînes std::string
F) Les conteneurs std::vector< >
G) Déclaration variables et portée
H) Alias de type, déclaration auto

Alias de type, déclaration auto

Alias de type, déclaration auto

- *En C++ les déclarations de types composés peuvent se transformer en tartines de code !*

C++

```
std::vector<std::vector<char>> faireGrille(size_t nbLig, size_t nbCol)
{
    std::vector<std::vector<char>> grille;

    /// Remplir la grille avec des push_back
    ...

    return grille;
}

int main()
{
    std::vector<std::vector<char>> petite = faireGrille(10, 15);
    std::vector<std::vector<char>> grande = faireGrille(20, 30);

    /// Utiliser les grilles...
```

Alias de type, déclaration auto

- *En C on avait typedef pour faire un alias de type...*
- *En C++ moderne (C++11) on préfère utiliser une déclaration d'**alias de type** avec* **`using`**

C++

```
using Grille = std::vector<std::vector<char>>;

Grille faireGrille(size_t nbLig, size_t nbCol)
{
     Grille grille;

    /// Remplir la grille avec des push_back


    return grille;
}

int main()
{
    Grille petite = faireGrille(10, 15);
    Grille grande = faireGrille(20, 30);

    /// Utiliser les grilles...
```

Alias de type, déclaration auto

- *Attention : ne pas rendre le code illisible pour les autres en introduisant plein d'alias mystérieux*
- *Préférer un nommage explicite, court et efficace*

C++

```
using VecVecChar = std::vector<std::vector<char>>;

VecVecChar faireGrille(size_t nbLig, size_t nbCol)
{
     VecVecChar grille;

    /// Remplir la grille avec des push_back


    return grille;
}

int main()
{
    VecVecChar petite = faireGrille(10, 15);
    VecVecChar grande = faireGrille(20, 30);

    /// Utiliser les grilles...
```

Alias de type, déclaration auto

- *Une alternative aux déclarations fastidieuses est la déduction automatique de type...*
- *Le type* **auto** *se déduit du contexte,* ***si possible !***

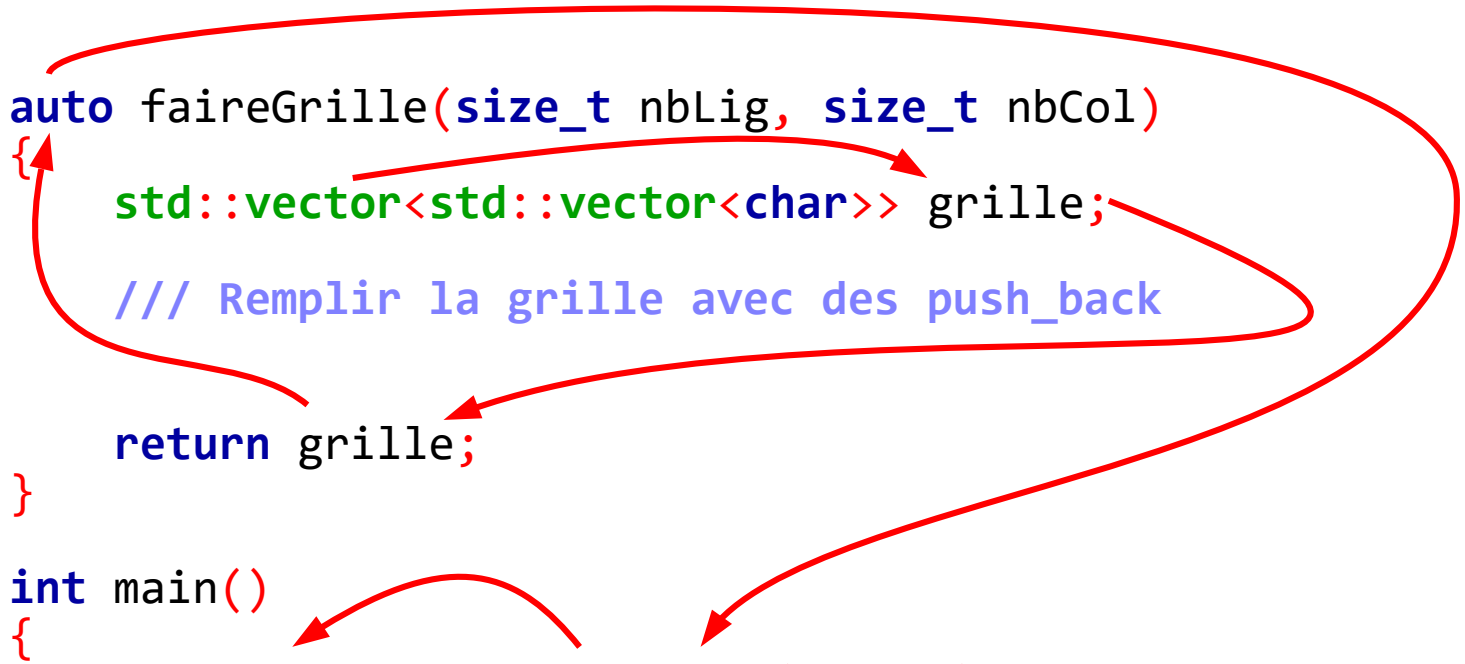
C++

```
auto faireGrille(size_t nbLig, size_t nbCol)
{
    std::vector<std::vector<char>> grille;

    /// Remplir la grille avec des push_back


    return grille;
}

int main()
{
    auto petite = faireGrille(10, 15);
    auto grande = faireGrille(20, 30);

    /// Utiliser les grilles...
```

Alias de type, déclaration auto

- *Une alternative aux déclarations fastidieuses est la déduction automatique de type...*
- *Le type* **auto** *se déduit du contexte,* ***si possible !***

C++



```
auto faireGrille(size_t nbLig, size_t nbCol)
{
    std::vector<std::vector<char>> grille;

    /// Remplir la grille avec des push_back

    return grille;
}

int main()
{
    auto petite = faireGrille(10, 15);
    auto grande = faireGrille(20, 30);

    /// Utiliser les grilles...
```

Alias de type, déclaration auto

- *Il doit y avoir un contexte, par exemple une initialisation dès la déclaration*
- *Le type* **`auto`** *se déduit du contexte,* ***si possible !***

C++

```
/// OK x sera de type int
auto x = 0;

/// OK y sera de type double
auto y = 1.0;

/// OK z sera de type float
auto z = 2.0f;

/// OK ch sera de type const char * (pas string)
auto ch = "Trois";

/// OK cond sera de type  bool
auto cond = false;
```

Alias de type, déclaration auto

- *Le compilateur fait ce qu'il peut mais ne peut pas prédire le futur ! Le C++ est **typé statiquement***
- *Un langage à typage dynamique saurait faire...*

C++

```
error: operands to ?: have different types 'int' and 'const char*'
auto w = cond ? 4 : "Quatre";
```

```
error: declaration of 'auto var' has no initializer
auto var;
if (cond)
    var = 5;
else
    var = "Cinq";
```