



INTRODUCTION AU C++ 14

Xavier Juvigny

18 Juin 2018

Prérequis et finalité du cours

Prérequis

- Avoir une bonne connaissance d'un langage de programmation
- Avoir si possible une bonne notion du langage C

Finalité du cours

- Connaître les différents paradigme de programmation
- Avoir une bonne notion d'une interface, de l'encapsulation et de la réutilisabilité
- Comprendre une classe C++ relativement complexe
- Comprendre un code écrit en template
- Avoir un aperçu des services proposés par la librairie standard du C++

Histoire du C++



1969 - Unix DEC PDP-7 - invention du langage **B** issu du langage **BCPL**



1972 - Portage d'Unix sur DEC PDP 11 –
Invention du langage **C**



1980 - première version du **C++**, dérivé du **C**, inspiré par **Simula67** et **Algol68**.

1998 - Première normalisation par l'ISO

2011- Deuxième normalisation

2014- Troisième normalisation

2017- Quatrième normalisation

2020-... : Une normalisation tous les trois ans de prévu...



Pour commencer...

Où chercher des tutoriaux et des conseils ?

Site du journal Dr Dobbs : <http://www.drdobbs.com/cpp>

Blog de Herb Sutter : <https://herbsutter.com/>

Guide de programmation C++

Guide de programmation communautaire supervisé par **Herb Sutter** et **Bjarn Stroustrup**, équivalent aux **PEPs** de **Python**

<https://github.com/isocpp/CppCoreGuidelines>

Sites de référence pour la norme ISO du C++ (98, 11, 14, 17, ...)

<http://en.cppreference.com/w/>

<http://www.cplusplus.com/>

Le C++ en quelques points

Le C++ est un langage :

- ✓ Compilé;
- ✓ Procédural;
- ✓ Structuré;
- ✓ Objet;
- ✓ Fonctionnel;
- ✓ Universel;

Le C++ est :

- ✓ Performant
- ✓ Parallèle (mémoire partagée)
- ✓ Utilisé par un grand nombre de programmeur.

Le C++ n'est pas :

- ✓ Un langage intuitif
- ✓ Rapide à maîtriser...

Compilateurs C++ usuels

Compilateur	Windows	Mac OS X	Linux
g++	✓	✗	✓
Clang++	✓	✓	✓
PGI c++	✓	✓	✓
cl (microsoft)	✓	✗	✗
icpc (intel)	✓	✗	✓

Aperçu du langage C++ : premières approches et production du code exécutable

Aperçu du langage

```
#ifndef _BONJOUR_HPP_  
#define _BONJOUR_HPP_  
#include <string>  
  
/** Fonction disant bonjour à <nom>  
    Input : <nom> : nom de la personne  
*/  
void  
dit_bonjour( const std::string& nom );  
  
#endif
```

Fichier de déclaration : `bonjour.hpp`

```
#include <iostream>  
#include "bonjour.hpp"  
  
void  
dit_bonjour( const std::string& nom )  
{  
    std::cout << "Bonjour " << nom  
               << "." << std::endl;  
}
```

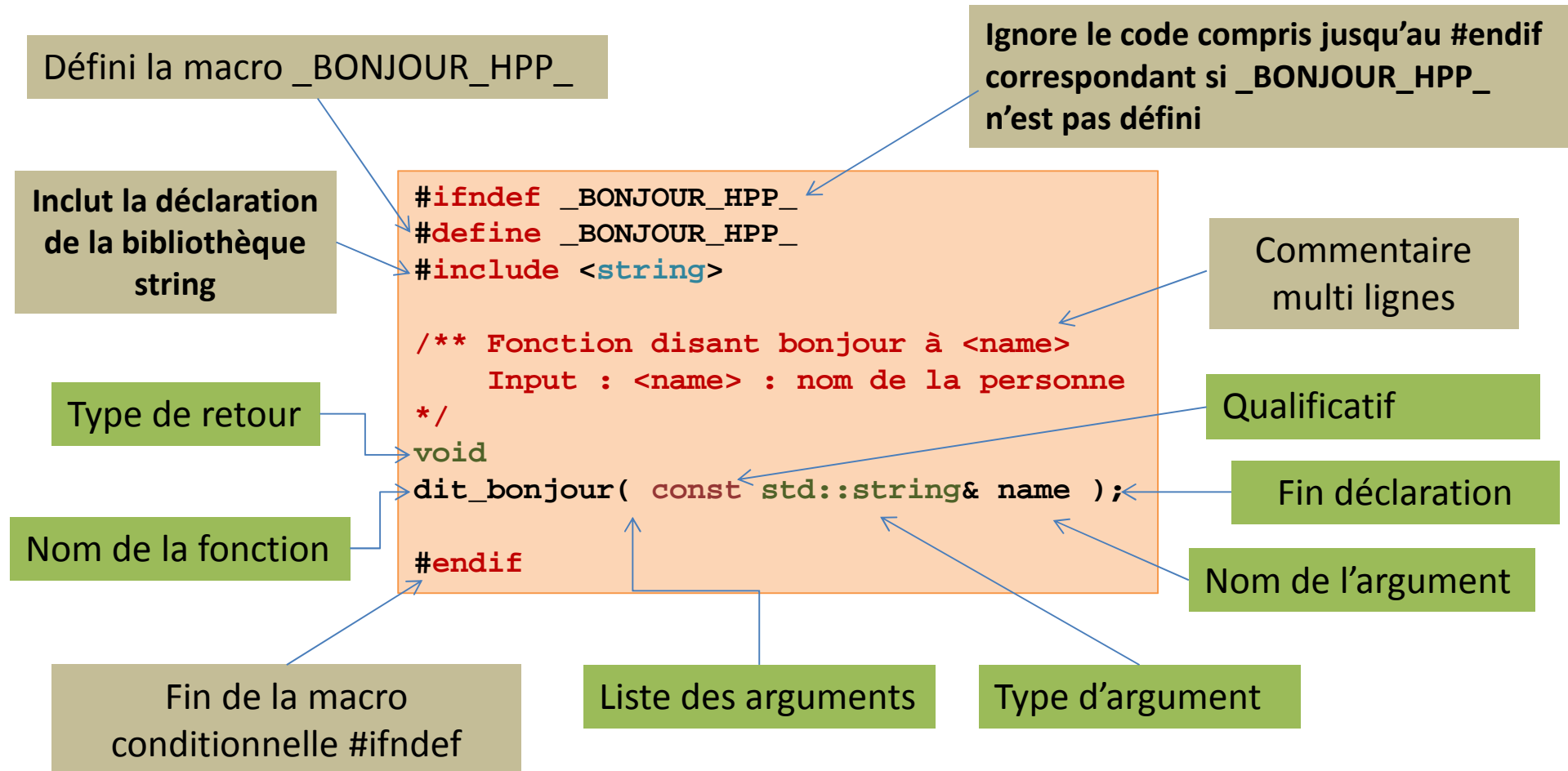
Fichier de définition : `bonjour.cpp`

```
#include "bonjour.hpp"  
  
// Programme principal.  
int main( int nargs, const char* argv[] )  
{  
    if ( nargs == 1 ) return EXIT_FAILURE;  
    dit_bonjour( argv[1] );  
    return EXIT_SUCCESS;  
}
```

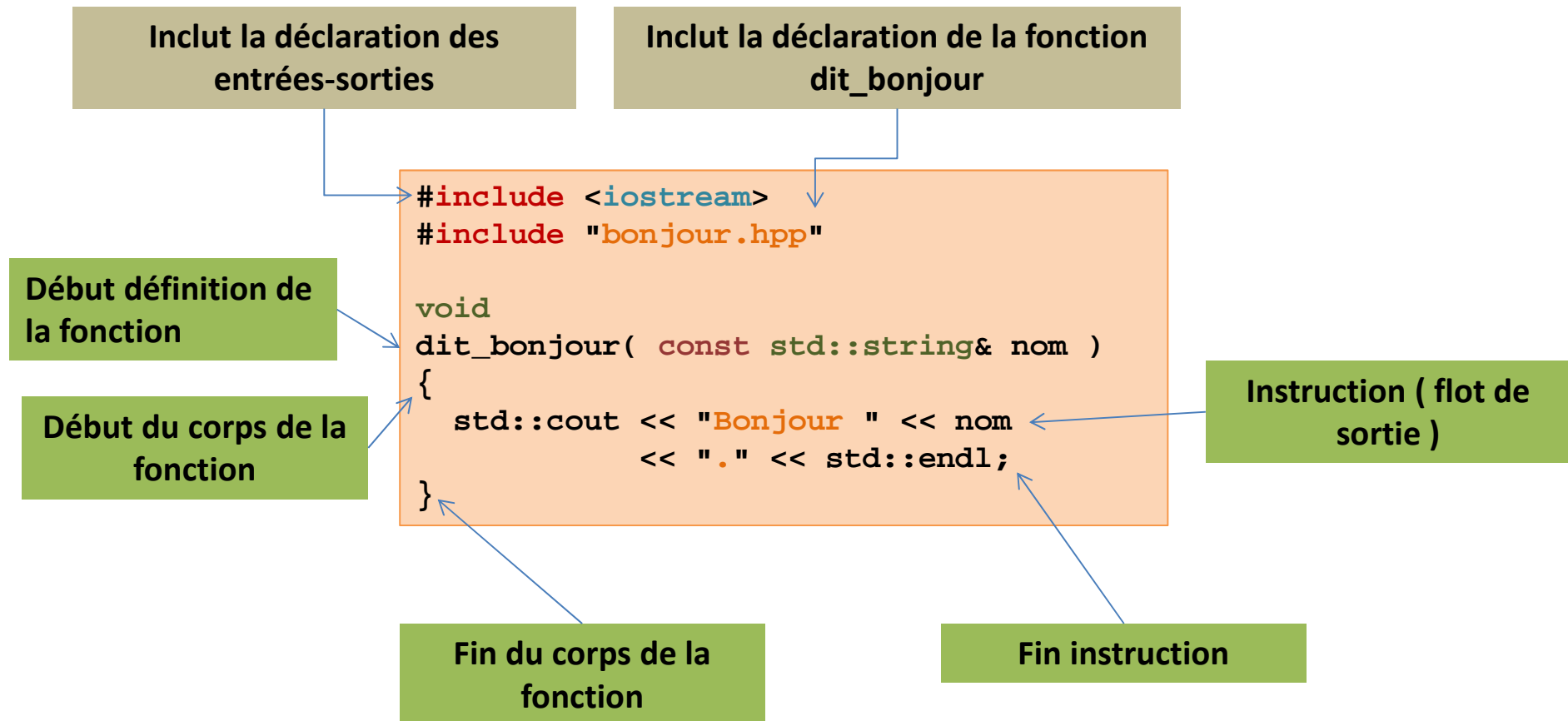
Fichier principal : `test_bonjour.cpp`

Le fichier de déclaration

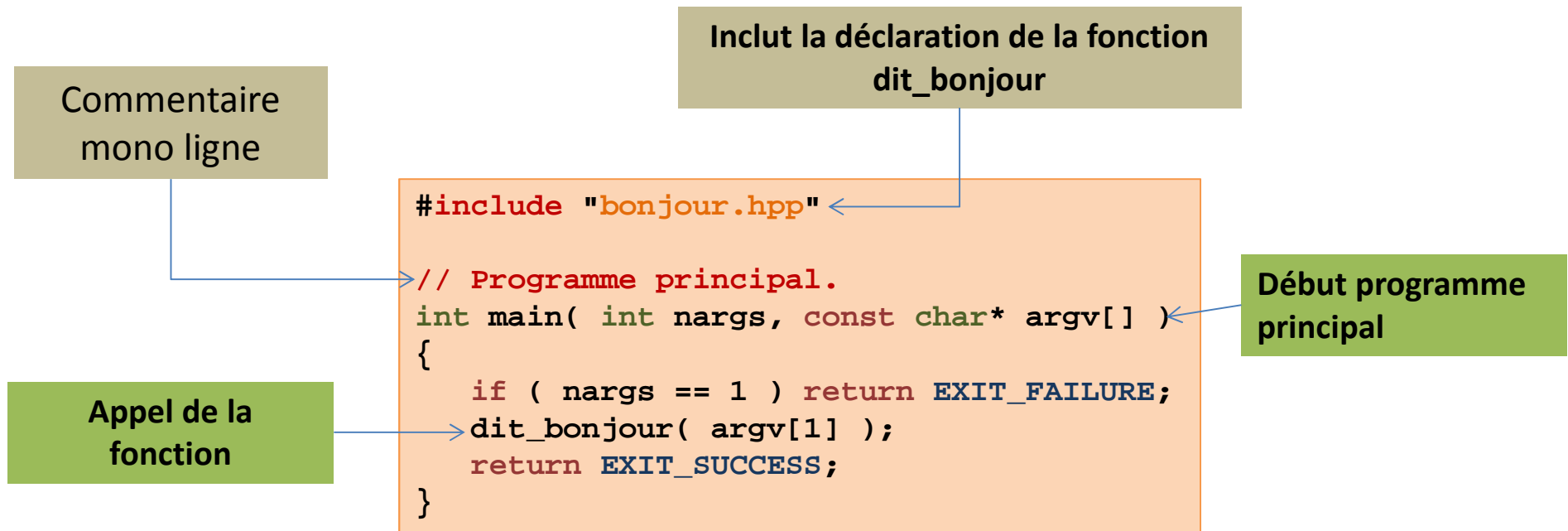
- Permet de déclarer une fonction : ce qu'elle retourne, et ce qu'elle attend en arguments
- Permet de définir de nouveaux types de variables
- On doit d'assurer qu'elle soit incluse **qu'une seule fois** dans un autre fichier !



Le fichier de définition



LE PROGRAMME PRINCIPAL



- Un seul programme principal (`main`) par programme
- Trois signatures possibles pour le `main` :
 - `int main()` : pas d'argument possible à l'exécution
 - `int main(int nargs, const char* argv[])` : possibilité arguments à l'exécution
 - `int main(int nargs, const char* argv[], char** envp)` : arguments + variables environnement (non standard, mais extension courante)

COMPILER LE PROGRAMME DIT_BONJOUR

- Choisir le compilateur utilisé pour créer l'exécutable (ici c++ !)
- Taper dans un shell la ligne de commande (sous unix, peut différer sous windows ou Mac OS X) :

```
c++ -o bonjour.exe -std=c++14 bonjour.cpp
```

- Pour compiler uniquement en C++ 2011, remplacer `-std=c++14` par `-std=c++11`
- Pour compiler uniquement en C++ 1998, remplacer par `-std=c++98`
- Par défaut, compile en C++ 2011 sur compilateurs récents, en C++98 sur des compilateurs un peu ancien.

Autres options utiles :

- Pour déboguer sereinement un programme, rajouter options : `-g -O0`
- Pour optimiser un programme : `-O3`
- Pour optimiser sur l'ordinateur local uniquement :
`-O3 -march=native` (avec g++/clang++)
`-O3 -xHost` (avec compilateur intel C++)
- Pour coller à la norme ISO : `-pedantic -Wall`
- Pour vérifier certaines règles recommandées dans la série de livre de Scott Meyer « Effective C++ » : `-Weffc++` (uniquement g++)

Pour visualiser la table des symboles d'un exécutable (ou autre binaire) :

```
nm -C bonjour.exe
```



DÉCLARATION DE VARIABLES EN C++ 14

DÉCLARATION DE VARIABLE EN C++

Syntaxe: `type nomvar [= valeur];`

- **Typage fort** : une variable ne peut en aucun cas changer de type en cours d'exécution
- Possibilité de déclarer sur une seule ligne plusieurs variables de même type :
`type nomvar1 [= valeur], nomvar2 [= valeur], ..., nomvar_n [= valeur];`

Exemple

```
int i=3, j, k = 2, l=-1;  
double x = 3.5, y = 2.7, z = 1.3;
```

- Interdiction de déclarer plusieurs fois la même variable dans un même bloc d'instruction

Exemple

```
{ // Début du bloc d'instruction  
  int i=3;  
  ...  
  int i = 2;      // Erreur, variable déjà déclarée  
  i = 4;          // OK, on change simplement sa valeur.  
} // Fin bloc d'instruction  
double i = 3.5; // Ok, pas même bloc d'instruction  
^              // donc pas même variable i...
```

RÈGLE DE VISIBILITÉ DES VARIABLES

- Une variable est visible dans son bloc d'instruction et les sous-blocs contenus ;
- Possibilité de déclarer une variable en dehors de tout bloc : **variable globale** (à éviter si possible)
- Deux variables ne peuvent pas avoir le même nom au niveau d'un bloc d'instruction ;
- Mais deux variables peuvent avoir le même nom si elles sont dans un bloc et un sous-bloc d'instruction ;
- Tant qu'une variable n'est pas déclarée dans un sous-bloc, le nom de variable utilisé fera référence à une variable déclarée dans un bloc contenant le sous-bloc;
- Une variable est détruite et son emplacement mémoire libéré à la sortie du bloc où elle a été déclarée.

Exemple

```
# include <iostream>
int i = 0; // variable globale

int main() { // Début du bloc d'instruction du programme principal
    std::cout << i << std::endl;
    int i = 1;
    std::cout << i << std::endl;
    { // Début d'un sous-bloc d'instruction
        int i = 2; // ::i fait référence au i global
        std::cout << i << " != " << ::i << std::endl;
        // Impossible par contre d'atteindre le i déclaré dans le main.
    } // Fin sous-bloc d'instruction
} // Fin bloc d'instruction du programme principal
```

VARIABLES ENTIÈRES

- Beaucoup de types d'entiers : codés sur 8, 16, 32 ou 64 bits, signés ou non...

Codage	8 bits	16 bits	32 bits	32/64 bits ⁽¹⁾	64 bits
Signé	signed char ⁽²⁾	short	int	long	long long
Valeurs admises	[-128 ; +127]	[-32768; 32767]	$[-2^{31} ; 2^{31} - 1]$	Selon 32/ 64 bits	$[-2^{63} ; 2^{63} - 1]$
Non signé	unsigned char	unsigned short	unsigned	unsigned long	unsigned long long
Valeurs admises	[0 ; 255]	[0 ; 65535]	$[0 ; 2^{32} - 1]$	Selon 32 / 64 bits	$[0 ; 2^{64} - 1]$

⁽¹⁾ : 32 bits sous Windows, 64 bits sous Linux

⁽²⁾ : char peut être signé ou non selon compilateur

Débordement d'entier si dépassement valeurs admises : C++ fait un modulo automatique...

Exemple

```
#include <iostream>

int main() {
    signed char c1 = -126, c2 = 126;
    c1 = c1 - 10; c2 = c2 + 10;
    // c1 --> 120, c2 --> -120
    std::cout << "c1: " << c1 << " c2: " << c2
               << std::endl;
    return EXIT_SUCCESS;
}
```



VARIABLES BOOLÉENNES

- Arithmétique basée sur la logique booléenne;
- Variable ne pouvant prendre que deux valeurs : **true** ou **false**;
- Valeurs équivalentes en entier : **true** → 1, **false** → 0, $\neq 0 \rightarrow \text{true}$, $0 \rightarrow \text{false}$
- A l'affichage, par défaut, affiche valeur entière (`std::noboolalpha`);
- Utiliser `std::boolalpha` (`iomanip`) pour afficher **true** ou **false**;
- Opérations logiques possibles pour booléens : par symbole ou en toute lettre

Symbole	&&		^	!
Opérande	and	or	xor	not

Exemple

```
#include <iostream>
#include <iomanip>
int main {
    bool b1, b2;
    b1 = true; b2 = false;
    bool b3 = b1 | b2, b4 = b1 or b2; // même opération pour b3 et b4
    std::cout << " b3 : " << b3 << " et b4 : " << b4 << std::endl;
    bool b3 = b1 & b2, b4 = b1 and b2; // Idem
    std::cout << std::boolalpha << "b3 : " << b3 << " et b4 : " << b4 << std::endl;
    bool b5 = (3 < 2); // b5 vaut false
    return EXIT_SUCCESS;
}
```

Opérateurs logiques

Deux types d'opérateurs booléens : logiques et arithmétiques

- **Logique** : n'évalue que vrai ou faux. Peut ne se servir que d'une des deux valeurs pour l'évaluation.
- **Arithmétique** : Evaluation logique bit à bit, renvoie une valeur entière, se sert toujours des deux valeurs pour l'évaluation.

```
int i1 = 13; // i1 = 0b1101
int i2 = 9; // i2 = 0b1001
int i3 = i1 && i2; // i3 = 1 ( vrai )
int i4 = i1 & i2; // i3 = 8
int i5 = i1 | i2; // i5 = 13 ( vrai )
int i6 = i1 || i2; // i6 = 15
bool f1 = true, f2 = false;
bool f3 = f2 && f1; // Seul f2 est évalué
bool f4 = f2 & f1; // f1 et f2 sont évalués.
int i7 = (i1 = 8) && (i2 = 4); // i7=1, i1=8, i2 = 4...
int i8 = (i1 = 3) & (i2 = 6); // i7 = 2, i1 = 3, i2 = 6
```

VARIABLES ENTIÈRES (SUITE)

Possibilité de mieux contrôler la taille des entiers : bibliothèque `cstdint`

`int8_t`, `int16_t`, `int32_t`, `int64_t` : Entiers signés de taille 8, 16, 32 ou 64 bits

`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` : Entiers non signés de taille 8, 16, 32 ou 64 bits

Opérations arithmétiques usuelles :

Symbole	+	-	*	/	%
Opération	Addition	Soustraction	Multiplication	Division	Modulo

Opérations arithmétiques inplace :

Symbole	+=	-=	*=	/=	%=
Opération	Add	Soustr.	Mult	Div	Mod

Exemple

```
#include <stdint>

int main {
    int32_t i = 5, j = 3;
    i += j; // Addition inplace, idem que i = i + j
    int64_t k = 103;
    k -= i; // Promotion automatique de i en int64_t durant l'opération
    return EXIT_SUCCESS;
}
```

PRÉ/POST INCRÉMENTATION/DÉCRÉMENTATION

- Possibilité d'incrémenter ou décrémenter de 1 une valeur (entière mais non obligatoire) : symboles ++ ou --
- Pour chacun, deux variantes possibles :
 - Pré : incrémente de un la valeur et retourne la nouvelle valeur. Notation : ++i; --j;
 - Post : incrémente de un la valeur et retourne l'ancienne valeur. Notation : i++; j--;

Exemple

```
#include <cstdint>
#include <iostream>

int main {
    int32_t i = 5, j = 3;
    int32_t k = i++, l = --j;
    std::cout << " k : " << k << " et l : " << l << std::endl;
    // Le programme devrait afficher :
    // k : 5 et l : 2
    return EXIT_SUCCESS;
}
```

En général : pré incrémentation/décrémentation **plus efficace** que la version post :
Pas besoin de conserver l'ancienne version...

TYPES RÉELS

Trois types de réels possibles :

- **float** : Réels simple précision (4 octets) :
 - Erreur relative $\varepsilon \leq 10^{-5}$, $\min \leq 10^{-37}$, $\max \geq 10^{+37}$
- **double** : Réels double précision (8 octets) :
 - Erreur relative $\varepsilon \leq 10^{-9}$, $\min \leq 10^{-37}$, $\max \geq 10^{+37}$
- **long double** : Réels long double précision (≥ 8 octets) :
 - Erreur relative $\varepsilon \leq 10^{-9}$, $\min \leq 10^{-37}$, $\max \geq 10^{+37}$

Attention : les valeurs données ici sont les valeurs minimales pour satisfaire le standard...

Opérations arithmétiques standards + fonctions usuelles dans bibliothèque **cmath**

Exemple

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

int main {
    double u0 = 1.;
    double u1 = std::sin(u0);
    double u2 = std::sin(u1);
    double diff = u1 - u2;
    std::cout << " diff : " << diff << std::endl;
    return EXIT_SUCCESS;
}
```

TYPES COMPLEXES

- Type non natif, on doit inclure une bibliothèque : `complex`
- Le type sous-jacent de la partie réelle et imaginaire choisi par le programmeur

Exemple

```
std::complex<double> z1; // Partie réelle et imaginaire de z1: réels double
std::complex<float>  z2; // Partie réelle et imaginaire de z2: réels float
std::complex<int>    iz3; // Partie réelle et imaginaire de z3: entier 32 bits
```

- Opérations arithmétiques usuelles + fonctions usuelles complexes
- Accès à la partie réelle et imaginaire d'un complexe : `z1.real()` et `z1.imag()`
- Initialisation d'un complexe entre `()` ou `{}` (à partir de C++ 11)

Exemple

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

int main {
    std::complex<double> z1(1.3,2.5); // z1 = 1.3 + 2.5i
    std::complex<double> z2{std::cos(1.45),std::sin(1.45)};
    std::complex<double> z3 = std::exp(std::complex<double>{0.,1.45}); // z3 = z2
    std::complex<double> zdiff = z3 - z2; // zdiff = 0. + 0.i
    std::cout << " ||zdiff|| : " << zdiff.norm() << std::endl;
    return EXIT_SUCCESS;
}
```

CHAÎNES DE CARACTÈRE

Deux types possibles :

- **POD** (Plain Old Data) issu du C : `const char*` `str`;
Uniquement utilisé pour des interfaces issus du C
- Avec type protégé (dans bibliothèque `string`) : `std::string` `str`;
A privilégié si possible, permet d'éviter un grand nombre d'erreur de programmation

Opérations possibles :

- Concaténation : `str3 = str2 + str1`; `str3 += str1`;
- Comparaison, insertion, recherche, etc.
- Conversion de valeurs réelles, entières en chaîne de caractère : `std::stoi`, `std::stod`, ...
- Codage ISO par défaut, mais possible de coder en `utf8`, `utf16` ou `utf32`
- Possibilité chaîne caractère « brute ». **Syntaxe** : `R"RAW(...)RAW"`
Où ... contient le texte (qui peut être multi lignes)

```
#include <string>
...
std::string str1("tin"), str2(" et "), str3("Pilou");
str1 += str1 + str2 + str3;
std::size_t pos_pilou = str1.find(str3); str1[pos_pilou] = 'M';
str1[0] = std::toupper(str1[0]);
int i = std::stoi("123"); double x = std::stod("3.14");
str1 = u8"chaînes avec des accents en utf8";
str2 = R"RAW(Voici une "chaine" avec quote
           et multiligne !RAW)";
str3 = u8R"RAW(Chaîne brute en utf8!!!RAW)";
```


ALIAS DE TYPE ET INITIALISATION DES VARIABLES

Possibilité de renommer un type à l'aide du mot clef **using**

Exemple : **using** dcomplex = **std::complex<double>**;

Initialisation des variables compatible C++ 98

```
int i = 4, j = 2;           // Initialisation avec symbole =
double x1(1.4), y1(1.5);    // Initialisation avec parenthèses
unsigned long ul = 9834534u; // Le u en fin du nombre spécifie un entier non signé
std::size_t addr = 0xFFA2045C; // Ecriture hexadécimale
dcomplex z1(0.5,1.3);       // Seule écriture possible
```

Initialisation des variables compatible C++ 11 ou supérieur

```
int i{4}, j{2};           // Initialisation avec accolades
dcomplex z1{0.5,1.3};
```

Initialisation des variables compatible C++ 14 ou supérieur

```
unsigned long u3 = 9'834'541u; // Ecriture entier avec séparateur
std::size_t addr{0xFF'A2'04'5C}; // Idem avec écriture hexadécimale
double ex = 3'432.4123'6543; // Idem avec réel ( dont séparateur après virgule )
// Ecriture binaire :
std::size_t addr2 = 0b11111111'10100010'00000100'01011110;
// nouvelle initialisation complexe :
using namespace std::complex_literals; // Indispensable en début de fichier
dcomplex z3 = 0.5 + 1.2i;
```

Branchements conditionnels

- Deux formes de branchement conditionnel : if ...else... et switch(...) case....

Branchement conditionnel `if (cond) { instructions... } [else { instructions... }]`

```
if ( n == 0 ) n = 10;  
if ( p < 0 ) { p = -p+1; } else { p = p + 1; }  
if ( double delta = b*b-4*a*c; delta < 0 )  
    std::cout << "Pas de racines !" << std::endl;
```

Uniquement C++ > 17 !

Branchement switch ... case

```
switch(car) // car est un caractère ( de type char )  
{  
    case '+':  
        res = x + y; break;  
    case '*':  
    case 'x':  
        res = x*y; break;  
    ...  
    case default:  
        std::cerr << "Opération inconnue !" << std::endl;  
}
```

Equivalent à '*' ou 'x'

Par défaut si pas de conditions préalables satisfaites

EXERCICE (15MN)

Résoudre l'équation de second degré : $a.x^2 + b.x + c = 0$

- Dans un premier temps, a,b et c, trois réels double précision, seront donnés dans le programme principal;
- Dans un second temps, on lira les valeurs de a,b et c à partir des arguments donnés à l'exécution du programme à l'aide de la fonction `std::stod`
- Dans le cas où le discriminant réduit est négatif, résoudre l'équation dans le corps complexe...
- Afficher la ou les solution(s) sur le terminal.

Qualifieurs C++

- Qualifieur : rajoute une qualité à une variable
- Types de qualifieurs en C++ : const, static, volatil

Qualifieur const

```
const int n = 3;  
const int m = n + 3;  
n = 2; ←  
const double x = 4;
```

Erreur à la compilation, n
est constante !

- Une valeur constante n'existe qu'à la compilation
- Possibilité d'opérations pour calculer une valeur constante

Qualifieur static

```
static int var_loc = 4;
```

- Variable visible que pour le fichier objet créé
- Mais dans d'autres contextes, autres significations qu'on verra plus tard

Qualifieur volatil

```
volatile int key_lock = 0;
```

- Variable ne pouvant être mis en cache
- Sert uniquement dans un contexte multithreadé (voir la fin du cours !)

Les références en C++

- Peut être vu comme un renommage d'une variable
- Nouvelle variable partageant la même valeur qu'une autre variable (même espace mémoire);
- Une référence est déclarée à l'aide du symbole &
- Une variable déclarée en référence doit obligatoirement être initialisée si dans un bloc d'instruction;
- On peut déclarer des paramètres en référence pour une fonction (voir plus loin)

<code>int i = 3.5;</code>	←	i vaut 3.5
<code>int &ref_i = i;</code>	←	ref_i prend la même valeurs que i
<code>int j = i;</code>	←	j vaut 3.5
<code>ref_i = 3;</code>	←	ref_i et i valent 3, j vaut 3.5
<code>j = 4;</code>	←	ref_i et i valent 3, j vaut 4



Typage automatique implicite des variables

- On laisse le compilateur déduire de lui-même le type de la variable
- Utile voire nécessaire mais ne pas trop en abuser sous peine de rendre le code non lisible ou pire avoir des bogues;
- Une fois qu'une variable est déclarée avec auto, son type est calculé par fois pour toute et ne peut pas varier au cours de l'exécution.

auto x= 3.5; ←

auto i = 3; ←

auto y = x +i; ←

auto j = 3, k = 4, l = j/k; ←

auto& y = x; ←

auto const& z = x; ←

auto w; ←

i = 3.14; ←

x sera déclarée comme double

i sera déclarée comme entier (int)

y sera déclarée comme double

j, k, l seront entiers, et l vaudra zéro !

y sera une référence sur x (donc sur un double)

z sera une référence constante sur x

Erreur ! Impossible de déduire le type de w

i vaudra 3 car de type entier !

Typage automatique explicite des variables

- Typage déduit à l'aide d'une formule donnée pour le typage mais non obligatoirement exécutée durant l'exécution du programme;
- Mêmes propriétés sinon que le typage automatique implicite

decltype(1+3.5) x;	←	x est de type double
decltype(x+(1.+3.i)) z;	←	z est de type complex<double>
decltype(std::norm(z)) y;	←	y est de type double
decltype((y)) ref_y = y;	←	ref_y est de type référence constante double sur y

Tableaux en C++

- Deux types de tableaux : **statique** et **dynamique**
- Tableau statique : Tableau dont la taille est connue et fixée à la compilation
- Tableau dynamique : Tableau dont la taille sera précisée à l'exécution et qui peut varier en cours d'exécution.
- En C/C++, que des tableaux à un seul indice de type entier (long, court, etc...).

Tableau statique

- Nombre d'éléments doit être connu et fixé à la compilation;
- Pas de temps d'allocation durant l'exécution;
- Accès rapide en lecture et écriture aux données du tableau.

Tableau dynamique

- Le nombre d'éléments peut ne être connu que durant l'exécution;
- Le nombre d'éléments peut varier en cours d'exécution;
- Temps d'allocation non négligeable;
- Accès aux éléments en lecture/écriture plus lent qu'avec un tableau statique.

Tableau statique

- Deux types de tableau statique : **P.O.D** (**P**lain **O**ld **D**ata) ou **staticarray**
- **P.O.D** : Tableau statique à la C
- **staticarray** : tableau statique C++ avec plus de contrôle possible pour les accès.

Tableaux statiques P.O.D

- Tableau statique de type C
- Simple mais pas de contrôle pour les accès : dépassement d'indices possible

```
double coord[3]; // Tableau de trois réels
```

```
double matrice[3][3]; // Tableau deux dimensions
```

```
coord[0] = 1.; coord[1] = 2; coord[2] = 3.;
```

```
matrice[0][0] = 1.; matrice[0][1] = 0.;
```

```
int permutation[3] = { 1, 0, 2 };
```

```
double mat_perm[3][3] = { { 0., 1., 0. },  
                          { 1., 0., 0. },  
                          { 0., 0., 1. } };
```

```
double x = mat_perm[0][3]; // <-- Erreur non détectée ni à la compilation ni à l'exécution
```

Tableau statique C++ (2011)

- Peut permettre un meilleur contrôle à l'accès aux données;
- Aussi performant que les tableaux statiques C

Initialisation tableau statique

```
#include <array>
```

Header à inclure en tête de fichier

Diverses initialisation

...

```
std::array<double,5> v, w = {1.,2.,3.,4.,5.}, z{-1.,-2.,-3.};
```

```
std::array<int,3> ip = {1,2,3,4};
```

```
std::array<std::array<double,3>,>3> matrice; // Matrice dimension 3 x 3
```

Erreur à la compilation

Accès

Tableau de tableau...

```
double x = v[3], y = v[4], z = v[5];
```

Bogue accès z, non détecté

Erreur compilation pour c

```
double a=std::get<3>(v),b=std::get<4>(v),c=std::get<5>(v);
```

```
int i = 3; double d = std::get<i>(v);
```

Erreur compilation indice non constant

```
const int j = 3; double e = std::get<j>(v); // OK, j est une constante
```

```
i = 5; double f = v.at(i);
```

Erreur traitable (exception)
lancée durant l'exécution

Efficacité : `std::get<i>(v) > v[i] > v.at(i)`

Tableau statique C++ (2011)

Autres accesseurs

`double x = v.front();`

Renvoi premier élément

`double z = v.back();`

Renvoi dernier élément

`std::size_t n = v.size ();`

Renvoi nombre d'éléments du tableau

`double* addr = v.data();`

Adresse premier
élément du tableau

Opérations disponibles

`v.fill(3.14); ip.fill(0);`

Rempli v avec valeur 3.14 et ip avec 0

`v.swap(w);`

Echange les valeurs de v avec celles de w

Opérations de comparaison

`bool b = (v==w);`

`b = (v<w);`

`b = (v<=w);`

`b = (v > w);`

`b = (v >= w);`

`b = (v != w);`

- Comparaison lexicographique/membre à membre
- Les tableaux doivent être de même taille !
- Opérateurs de comparaison possible si les éléments contenus sont comparables
- Complexité linéaire avec la taille

Tableau dynamique en C++ (1998 et +)

- Permet d'allouer et désallouer un tableau en cours d'exécution du programme;
- Temps d'allocation non négligeable (comme toute allocation dynamique);
- Plus souple d'utilisation que les tableaux statiques, swap plus efficace.
- Notion place allouée différente de nombre d'éléments du tableau.

Initialisation d'un vecteur dynamique

```
#include <vector>
```

Bibliothèque à include avant toute utilisation de vector

```
...
```

```
std::vector<double> u;
```

Création tableau de zéro élément (vide)

```
std::vector<double> v(10);
```

Création tableau 10 éléments

```
int n =5; std::vector<double> w(n);
```

Création de n éléments initialisés à 0

```
std::vector<int> ind(n, 0);
```

```
std::vector<double> e1{1.,0.,0.};
```

Création tableau de 3 éléments initialisés (2011)

Accès aux éléments

```
double x = v[0], xn = v[11];
```

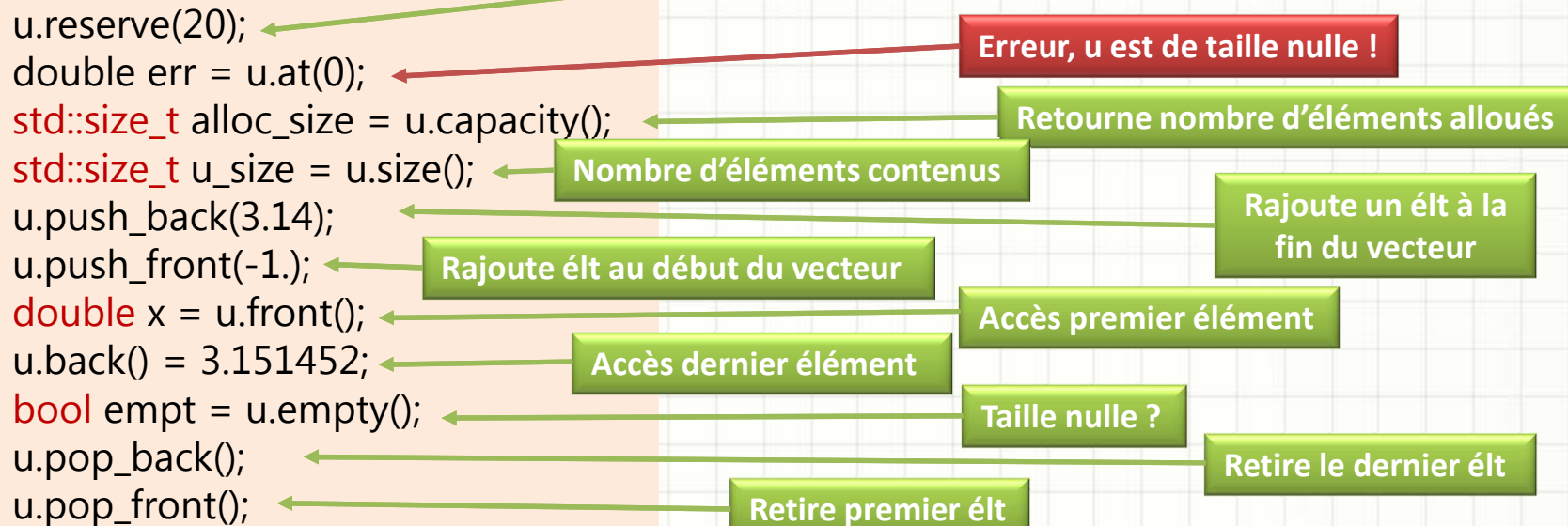
Accès sans contrôle. Bogue pour xn

```
x = v.at(0); xn = v.at(10);
```

Accès avec contrôle. Erreur levée pour xn

Tableau dynamique C++ (98 et +)

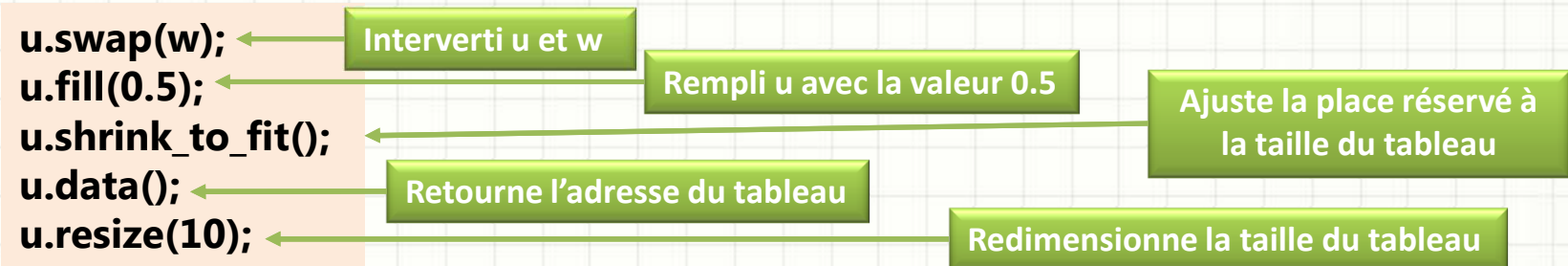
Accesseurs/Modifieurs



- Si assez de mémoire réservée, `push_back` ne fait pas d'allocation
- Retirer un élément avec `pop_back` conserve la place mémoire allouée
- Combiner `reserve` avec `push_back` permet de rajouter très efficacement des éléments à la fin d'un tableau
- En revanche, `push_front` et `pop_front` demande de déplacer des données en mémoire (complexité en $O(n)$)

Tableau dynamique (C++ 98 et +)

Autres fonctionnalités



- L'échange de tableau avec **swap** est très efficace. Performant même pour realloquer un tableau en perdant ses données !
`std::vector<double>(20).swap(u);` // Realloue 20 éléments pour u
- **shrink_to_fit** realloque et copie les éléments du tableau (**uniquement C++11 ou +**)
- Le redimensionnement du tableau avec **resize** très efficace si la taille demandée est inférieure à la taille allouée, sinon déclenche une reallocation avec copie des éléments.

Autres accesseurs/modifieurs

```
std::vector<std::vector<double>> matrix(3);  
matrix.reserve(3);  
matrix.emplace_back(std::vector<double>{1.,0.,0.});  
matrix.emplace_back(std::vector<double> {0.,1.,0.});  
matrix.emplace_back(std::vector<double> {0.,0.,1.});
```

Les tableaux avec GNU C++ (g++)

- Possibilité de déboguer l'accès aux tableaux statiques et dynamiques avec g++
- Rajouter l'option `-D_GLIBCXX_DEBUG` permet de vérifier l'accès aux éléments même sans utiliser la fonction `at()`;
- Rajouter l'option `-D_GLIBCXX_DEBUG_PEDANTIC` vérifie encore plus de choses lors de l'utilisation des bibliothèques standards du C++

Les boucles en C++

- Quatre types de boucles en C++ :
 - Les boucles de type **do { ... } while (cond);**
 - Les boucles de type **while(cond) { ... }**
 - Les boucles de type **for (init ; cond; incrément) { ... }**
 - Les boucles de type **for (variable : conteneur) { } (C++11 ou +)**

Boucles `do { ... } while(cond);` et `while(cond) { ... }`

```
std::vector<double> u{1.,3.,2.,-4.,9.,1.};  
double x;  
do  
{  
    x = u.back();  
    u.pop_back();  
} while (x>0);
```

En sortie : `u = { 1., 3., 2. };`

```
std::vector<double> u{1.,3.,2.,-4.,9.,1.};  
double x = u.back();  
while (x>0)  
{  
    u.pop_back();  
    x = u.back();  
}
```

En sortie : `u = { 1., 3., 2.,-4. };`

La boucle for (init; cond; incrément) en C++

- Boucle très généraliste et puissante en C (et C++)
- On peut y initialiser plusieurs variables comme en C
- Possibilité de déclarer une variable dans la partie d'initialisation, mais portée de la variable uniquement dans le corps de la boucle
- L'incrément accepte n'importe quelle expression arithmétique !

```
std::vector<int> ui{3,6,9,12,15,18,21,24}, vi{7,14,21,28,35};  
int val = -1;  
for ( int i = 0, j = 0; (i < ui.size()) and (j < vi.size()) and (ui[i] != vi[j]); i++,j++)  
{  
    val = ui[i];  
}
```

Initialisation Condition de continuité Incrément(s)

```
for ( val = 3072; val %2 == 0; val /= 2 );  
val = 135;  
for ( ; val%3 == 0; val /= 3 );
```

La variable i n'est visible que dans sa boucle, tandis que val est de portée globale.

La boucle for (var : conteneur) en C++11

- Permet de parcourir en lecture/écriture les éléments d'un conteneur (tableaux entre autre);
- La variable déclarée dans la boucle peut soit prendre successivement toutes les valeurs contenus (pour la lecture) soit y référer (voir plus loin) pour l'écriture et lecture.

```
std::vector<double> u{1.,3.,5.,7.,9.};  
double sum = 0.;  
for ( auto val : u ) sum += val;  
double x = 0;  
for ( auto& val : u )  
{  
    double y = x;  
    x = val;  
    val = val + y;  
}
```

val prend successivement toutes les valeurs contenues dans u.

val « devient » successivement chaque valeur contenue dans u

Ici, on modifie bien des valeurs stockées dans u !

- Le type de val sera déduit par le compilateur selon le type contenu dans u;
- Val est de portée locale dans chaque boucle
- Ce type de boucle ne se limite pas qu'aux tableaux, mais pourra être utilisé pour les listes C++, des dictionnaires, ... et même des types définis par le programmeur.

Les conversions de type

- A faire...

Exercices récapitulatifs

Nombres impairs

Ecrire un programme stockant les nombres impairs en mémoire (au moins trois solutions);

Crible d'Erasthote

Ecrire un programme qui calcule les nombres premiers compris entre 2 et un entier n choisi par l'utilisateur, en utilisant le crible d'Erasthote :

- Construire un tableau contenant tous les nombres entiers compris entre 2 et n
- Supprimer tous les multiples de 2 en les mettant à la valeur nulle;
- Puis supprimer les multiples du prochain entier non encore supprimé par les étapes précédentes et recommencer jusqu'à atteindre un entier supérieur à la racine carrée de n .

Autres conteneurs proposés par le C++

Les listes doublement chaînées

- Moins performantes que vector
- A n'utiliser que si on doit faire beaucoup d'insertion dans la liste...

```
#include <list>
```

```
...
```

```
std::list<int> li{1,2,5,7};  
li.push_back(9); li.pop_front();  
li.pop_back(); li.push_front(-1);  
li.emplace_back(2);li.emplace_front(1);  
for ( auto v : li ) std::cout << v << " ";
```

Les ensembles ordonnés

- Valeurs uniques dans l'ensemble
- Existe en version non ordonnés

```
#include <set> // unordered_set sinon
```

```
...
```

```
std::set<int> si{1,2,5,7};  
si.insert(3); si.insert(5);  
for ( auto v : si ) std::cout << v << " ";
```

Les dictionnaires ordonnés

- Dictionnaire : clef + valeur
- Existe en non ordonné
- Clef de type comparable, si entier, indices pas nécessairement continus

```
#include <map> //unordered_map else
```

```
...
```

```
std::map<std::string,int> di;  
di["hector"] = 843404;  
for ( auto v : di )  
std::cout << v.first  
           << " -> " << v.second  
           << " ";
```

Autres conteneurs

Les paires

```
#include <utility>
...
std::pair<long, std::string> badge_id;
badge_id.first = 1664;
badge_id.second = "François";
std::pair<std::string, std::vector<double>> named_vect =
    {"Magnetic field", std::vector<double>{1., 2., 3., 4.}};
```

Les tuples

```
#include <tuple>
...
auto tples = std::make_tuple("Energy",
                             std::vector<double>{1., 2., 3.4.},
                             3.1415, 10);
std::string name = std::get<0>(tples);
double time;
std::vector<double> field;
int id_block;
std::tie(std::ignore, field, time, id_block) = tples;
std::size_t sz = std::tuple_size<decltype(tples)>::value;
```


Exercices sur les conteneurs

Statistiques sur une chaîne de caractères

- Compter le nombre de minuscules, de majuscules, de chiffres ou d'autres caractères;
- Afficher en fin de programme les caractères trouvés en un seul exemplaire chacun

Conseil : Regarder sur internet la documentation de la librairie `cctype`

Suite de Syracuse

On considère la suite de Syracuse :

$$u_n = 3u_{n-1} + 1 \text{ si } u_{n-1} \text{ est impair; } \frac{u_{n-1}}{2} \text{ si } u_{n-1} \text{ est pair}$$

Avec u_0 choisi

- Ecrire un programme qui fait varier u_0 de 5 à 10000 et qui itère pour chaque u_0 jusqu'à tomber sur la valeur 1;
- Calculer la valeur maximale et le nombre d'itérations de la suite;
- Conserver les deux suites représentant celle ayant atteint la plus grande valeur possible et celle ayant itéré le plus;
- Afficher les deux suites.

Pointeurs en C++

0xFF AE 08 4A 9B 0F CA 04 → 0x0F 0x00

```
int16_t *pt_i = &i;
```

```
int16_t i = 255;
```

- Variable entière longue non signée désignant une case de la mémoire virtuelle;
- Plusieurs types de pointeurs :
 1. **Pointeur C** (**Plain Old Data** type) : Simple entier long non signé pouvant pointer ou non sur une variable.
 2. **Pointeur unique** : Un seul pointeur de ce type peut pointer simultanément sur une valeur donnée. La valeur est détruite dès qu'aucun pointeur ne pointe dessus.
 3. **Pointeur partagé** : Plusieurs pointeurs peuvent pointer sur une même valeur qui sera détruite dès qu'aucun pointeur pointe dessus.

Une valeur de pointeur particulière : le pointeur `nullptr`

- Garantit de ne pointer sur aucune valeur en particulier
- Possède un type de valeur particulier `nullptr_t` convertible en n'importe quel autre type de pointeur

P.O.D pointer (Pointeur C)

Déclaration

```
double x = 3.14, y = 1.5, z = 2.1;
int     i   = 2;
float   tab[] = { 1.0f, 2.0f, 3.0f };
void    *ptr = nullptr; // void : pointeur non typé...
double *pt_x=&x, *pt_y=&y, *pt_z=&z, *pt_w=nullptr;
int     *pt_i = &i;
float   *pt_tab = tab;
```

Accès à la valeur pointée

```
*pt_x = 3.15; // Maintenant, x vaut 3.15 !
double w = (*pt_y) + (*pt_z); // w vaut 3.6
*ptr = 2; // Erreur, ptr pointe sur nullptr !
pt_tab[1] = -2.0f; // Le 2e élt de tab devient -2.0f
pt_z[0] = 2.2; // z vaut maintenant 2.2
pt_z[1] = 3.3; // Erreur, pt_z ne pointe que sur une
               // variable, pas un tableau
```

Pointeurs P.O.D

Arithmétique de pointeur

```
std::vector<double> u{1.,2.,3.,4.,5.,6.,7.,8.,9.};  
double* pt_u = u.data();  
pt_u += 5; double v = *pt_u;  
pt_u -= 2; v = *pt_u;  
Double* pt_v = pt_u + 5;  
Long distance = pt_v - pt_u;  
Double* pt_w = pt_u++;  
Double* pt_x = --pt_v;
```

pt_u pointe sur le premier élt de u
pt_u pointe maintenant sur le 6^e élt
pt_u pointe maintenant sur le 4^e élt
pt_v pointe sur le 9^e élt de u
distance = 5 (5 élt entre u et v)
pt_w pointe 4^e élt, pt_u 5^e élt
pt_x et pt_v pointent le 8^e élt.

- Arithmétique puissante mais de bas niveau (proche de l'assembleur);
- Pas de garde fou pour les dépassements mémoire;
- Ne marche pas avec le pointeur non typé void*;
- Pointeurs P.O.D sont les seuls à pouvoir utiliser l'arithmétique de pointeur.

Pointeurs uniques en C++ 11

- Doit include l'entête de la bibliothèque memory;
- Permet de construire une valeur dynamiquement;
- Une seule variable peut pointer à la fois sur cette valeur;
- Quand plus aucune variable ne pointe dessus, la valeur est détruite;
- Pas d'arithmétique de pointeur pour ce type de pointeur.

```
#include <memory>
```

```
std::unique_ptr<double>un_pt;  
auto un_pt_v=std::make_unique<double>(4.56);  
std::unique_ptr<double>un_pt_w=std::move(un_pt_v);  
double a=un_pt_w[0];  
std::cout<<*un_pt_w;
```

Pointe sur valeur nulle

Pointe sur la valeurs 4.56

un_pt_w pointe sur 4.56,
un_pt_v sur nullptr;

Même interface que pointeur P.O.D

- Pas d'incrément/décément possible
- Un pointeur unique déclaré dans un bloc détruit la variable sur laquelle elle pointe à la fin de ce bloc.

Pointeurs partagés (C++ 11)

- Doit inclure le fichier d'entête <memory>;
- Permet de construire une valeur dynamiquement;
- Plusieurs pointeurs peuvent partager une même valeur;
- Lorsque plus aucun pointeur ne pointe sur cette valeur, elle est détruite;
- Possibilité de connaître le nombre de pointeurs partageant une valeur;
- Même interface que les pointeurs P.O.D mais pas d'arithmétique de pointeur.

```
#include <memory>
```

```
...
```

```
std::shared_ptr<double> un_pt;
```

Pointe sur nullptr

```
auto un_pt_v = std::make_shared<double>(4.56);
```

```
auto un_pt_w = un_pt_v;
```

un_pt_v et un_pt_w pointent sur la même valeur (4.56)

```
int ref_count = un_pt_w.use_count();
```

Renvoie 2

```
un_pt_v = std::make_shared<double>(4.56);
```

```
ref_count = un_pt_w.use_count();
```

Renvoie 1

```
un_pt_w = nullptr;
```

La première valeur est détruite

La seconde valeur est automatiquement détruite à la fin du bloc.

Les fonctions en C++

- Prend des arguments pouvant être éventuellement modifiés en entrée;
- Retourne une valeur ou non (retourne void si aucune valeur retournée);
- Les arguments sont passés par défaut par copie;
- Les arguments passés en référence sont passés par adresse;
- Les arguments passés par référence ne peuvent pas être directement des valeurs sauf si ils sont passés en référence constante.

```
double f1( std::vector<double> u );
```

Déclaration de la fonction

```
...
```

```
double f1( std::vector<double> u ) {
```

Définition de la fonction

```
    double sum = 0.;
```

```
    for ( auto val : u )
```

```
        sum += val;
```

```
    u[0] = sum;
```

```
    return sum;
```

```
} ...
```

```
int main()
```

```
{
```

```
    std::vector<double> u{1.,2.,3.,4.};
```

```
    double s = f1(u);
```

Appel de la fonction

```
....
```

Fonction f1 non optimale :

- *u passée par valeur.*
- A l'appel de f1, on copie u dans un tableau temporaire utilisé ensuite par la fonction.
- u étant copié, ne sera pas modifié par la fonction f1.

Les fonctions en C++ : passage par référence

- Permet un passage par adresse d'un argument;
- Evite une copie pour un code plus optimal;
- Permet de modifier la valeur passée en argument.

```
double f2( std::vector<double>& u ) {  
    ...  
    u[0] = sum;  
    return sum;  
}  
double f3( const std::vector<double>& u ) {  
    ...  
    u[0] = sum;  
    return sum;  
}  
int main() {  
    std::vector<double> arr{1.,2.,3.,4.};  
    double s1 = f1(arr);  
    double s2 = f2(arr);  
    double s3 = f2(std::vector<double>{1.,2.,3.});  
    double s4 = f3(arr);  
    double s5 = f3(std::vector<double>{1.,2.,3.});  
}
```

Erreur compilation : u n'est pas modifiable (const)

arr non modifié car copié à l'appel

arr modifié à l'appel

Erreur compilation : pas une variable

Ok car référence constante

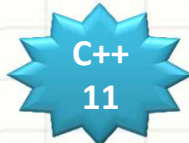
Syntaxes possibles pour les fonctions

Plusieurs syntaxes possibles pour déclarer ou définir des fonctions

Syntaxe classique

`type_retour nom_fonction(paramètres)`

Syntaxe fonctionnelle



`auto nom_fonction(paramètres) -> type_retour`

Syntaxe déductive



`auto nom_fonction(paramètres)`

```
double normL2( const std::vector<std::complex<double>>& u ) {  
    decltype(std::norm(u[0])) sum = 0.;  
    for ( const auto& v : u ) sum += std::norm(v);  
    return std::sqrt(sum);  
}
```

// Style C++11

auto

```
normL2(const std::vector<std::complex<double>>& u) -> decltype(std::norm(u[0]));
```

// Style C++14 déductif...

```
auto normL2( const std::vector<std::complex<double>>& u );
```

Fonction inline, fonction statique

Fonctions inline

- Fonction dont le code sera inliné, c'est-à-dire remplaçant l'appel à cette fonction
- Gain de temps en évitant un appel à une fonction
- Mais si fonction trop importante, taille du code, grossit et, boucles appelant cette fonction trop grosses pour rentrer dans le cache instruction du processeur.

```
inline std::array<double, 3> cossin(double x) {  
    return {std::cos(x),std::sin(x)};  
}  
inline double norm(const std::array<double,3>& u ) {  
    return (u[0]*u[0]+u[1]*u[1]+u[2]*u[2]);  
}
```

Fonctions statiques

Fonction invisible en dehors de sa zone de définition

```
static double compute_square_norm( int dim, const double* x) {  
    double s = 0; for ( int i = 0; i < n; ++i ) s += x[i];  
    return s;  
}  
double norm(const std::vector<double>& u )  
{ return compute_square_norm(u.size(), u.data()); }
```

Type de retour d'une fonction

par valeur

Deux modes de retour :

- Si la valeur retournée est une variable *passée en paramètre* ou *globale*, on retourne **une copie** de la valeur
- Si la valeur est une variable *locale*, on retournera **un déplacement** de cette valeur (équivalent à l'instruction **std::move**) : pour un vecteur, la variable recevant la valeur retournée « volera » le pointeur de cette dernière (pas de copie).

par référence

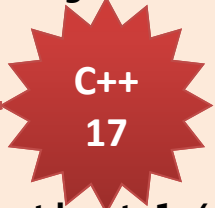
- Si la valeur est *globale* ou *passée en paramètre*, on retourne une référence sur cette valeur (pas de copie si la variable qui reçoit est une référence également);
- **Si la variable est locale, on retourne une valeur détruite à la sortie de la fonction : plantage du programme assuré !**

Retour multiple

- Possibilité de retourner plusieurs valeurs homogènes ou hétérogènes à l'aide des tableaux statiques, des paires et des tuples;
- **std::tie** et **std::get** permettent de récupérer ensuite ses diverses valeurs;
- En C++ 17, possible aussi de les récupérer au travers de l'opérateur [...]
- Permet la création de **fonctions pures** (voir programmation fonctionnelle).

Exemples de fonctions multi-retour

```
std::pair<int,int> div_and_mod( int i, int j ) {  
    return {i/j,i%j};  
}  
std::array<double,3> cos_sin_and_tan( double x ) {  
    return {std::cos(x), std::sin(x), std::tan(x)};  
}  
std::tuple<int,double> int_and_res( double x ) {  
    return std::make_tuple( int(x), x-int(x) );  
}  
...  
int main()  
{  
    auto res = div_and_mod(7,3);  
    std::cout << res.first << ", " << res.second << std::endl;  
    std::cout << std::get<0>(res) << ", " << std::get<1>(res) << std::endl;  
    int d,r;  
    std::tie(d,r) = div_and_mod(8,3);  
    auto [e,f] = int_and_res(3.1415);  
    auto [s,c,t] = cos_sin_and_tan(e);  
    auto res2 = int_and_res(1.414);  
    std::cout << std::get<0>(res2) << ", " << std::get<1>(res2) << std::endl;  
}
```



Surcharge des fonctions

Surcharge des fonctions

Même nom de fonction mais signature différente

```
int pow_n( int x, int n ) {  
    return ( n == 0 ? 1 : n%2==0 ? pow_n(x*x,n/2) : x*pow_n(x*x,(n-1)/2) );  
}  
double pow(int x, int n) { // Erreur compilation,  
    // même signature que fonction précédente  
    return ( n == 0 ? 1 : n%2==0 ? Pow_n(x*x,n/2) : x*pow_n(x*x,(n-1)/2) );  
}  
float pow_n( float x, int n ) {  
    return ( n == 0 ? 1 : n%2==0 ? pow_n(x*x,n/2) : x*pow_n(x*x,(n-1)/2) );  
}  
double pow_n( double x, int n ) {  
    return ( n == 0 ? 1 : n%2==0 ? pow_n(x*x,n/2) : x*pow_n(x*x,(n-1)/2) );  
}  
int main() {  
    int iy = pow_n(3,5);  
    float fy = pow_n(3.f, 5);  
    double y = pow_n(3.,5);  
    ...  
}
```

Paramètres par défaut

Paramètres optionnels, toujours définis en dernier, uniquement définis à la déclaration.

// Dans blas1.hpp

```
void axpy( int n, double a, const double* x, double* y, int incx = 1, int incy = 1 );
```

...

// Dans blas1.cpp

```
void axpy(int n, double a, const double* x, double* y, int incx, int incy ) {  
    for ( int i = 0; i < n; ++i ) y[i*incy] += a * x[i*incx];  
}
```

```
int main() {
```

```
    const int dim = 1000;
```

```
    std::vector<double> u(dim), v(dim);
```

...

```
    axpy( dim, 0.5, u.data(), v.data() );
```

$$v_i = v_i + 0.5 * u_i$$

```
    axpy( dim/2, 0.5, u.data(), v.data(), 2 );
```

$$v_i = v_i + 0.5 * u_{2i}$$

```
    axpy( dim/2, 0.5, u.data(), v.data(), 1, 2 );
```

$$v_{2i} = v_{2i} + 0.5 * u_i$$

```
    axpy( dim/2, 0.5, u.data(), v.data(), 2, 2 );
```

$$v_{2i} = v_{2i} + 0.5 * u_{2i}$$

Surcharge des opérateurs

Possibilité de redéfinir à l'aide de fonction certains symboles (+, -, *, (), [], etc.)

- Opérateurs unaires : un paramètre à droite ou à gauche du symbol
- Opérateurs binaires : premier paramètre à gauche du symbole, deuxième à droite
- Opérateurs n-aires : uniquement opérateur d'évaluation (...)

```
std::vector<double> operator + ( const std::vector<double>& u,  
                                const std::vector<double>& v ) {  
    std::vector<double> w(u.size());  
    for ( int i = 0; i < u.size(); ++i ) w[i] = u[i] + v[i];  
    return w;  
}  
std::vector<double> operator - ( const std::vector<double>& u ) {  
    std::vector<double> w(u.size());  
    for ( int i = 0; i < u.size(); ++i ) w[i] = -u[i];  
    return w;  
}  
int main() {  
    std::vector<double> u, v; ...  
    auto w1 = u+v;  
    auto w2 = -w1;
```

Surcharge des opérateurs arithmétiques et logiques

Opérateurs arithmétiques et logiques

- Arithmétiques **unaire** : -, **binaire** : +, -, *, /
- Logiques : **booléen** : &&, ||, !, **arithmétiques**: &, |, ^

Remarque : les symboles peuvent être redéfinis pour n'importe quels types.

Opérateurs arithmétiques et logiques inplace

- Arithmétiques : +=, -=, *=, /=
- Logiques : &=, |=, ^=

```
std::vector<double>&  
operator += ( std::vector<double>& u, const std::vector<double>& v ) {  
    for ( std::size_t i = 0; i < u.size(); ++i ) u[i] += v[i];  
    return u;  
}  
...  
int main () { ...  
    u += v;  
    ...
```


Exercices sur les fonctions

Orthonormalisation d'un ensemble de vecteurs

On veut générer une base orthonormale à partir d'un ensemble de fonctions à l'aide d'un algorithme de Gram-Schmidt modifié, dont l'algorithme a été mis en œuvre dans le programme principal, ainsi que la génération de la famille libre U .

Complétez le programme (sans modifier le main !) en se reposant sur les spécifications induites par le programme principal.

Les structures

Crée de nouveaux types de données composées d'autres types

Syntaxe

```
struct nom_struct {  
    type1 nom_champs_1;  
    type2 nom_champs_2;  
    ...  
    type_ret nom_fonction(args...) [const];  
    ... };
```



- Possibilité d'associer des fonctions (méthodes) traitant les données d'une valeur du type de la structure (voir chapitre sur la programmation objet);
- Possibilité d'initialiser tous les champs en un coup

```
struct employe {
    std::string nom, prenom, adresse;
    unsigned long long id_badge, id_cpam;
};

...
employe r_chambier = { "Chambier", "Robert",
                       u8"4 rue des batteries, 92100 Châtillon",
                       100438ULL, 16875432403ULL};
```

Les structures

Deux modes d'accès aux champs : via une valeur et via un pointeur

```
void display( const employe& e ) {
    std::cout << "Nom : " << e.nom << ", prenom : " << e.prenom << "\n"
               << e.adresse << "\n" << "Badge entreprise : " << e.id_badge
               << "\nnum. secu. soc. : " << e.id_cpam << std::endl;
}

std::shared_ptr<employe>
make_employe( const std::string& prenom, const std::string& nom,
              const std::string& adresse, unsigned long long id_cpam,
              unsigned long long id_badge ) {
    auto pt_employe = std::make_shared<employe>();
    pt_employe->prenom = prenom; pt_employe->nom = nom;
    pt_employe->adresse = adresse; pt_employe->id_cpam = id_cpam;
    pt_employe->id_badge = id_badge;
    return pt_employe;
}

int main() {
    auto pt_robert = make_employe( "Robert", "Chombier",
                                   "4 rue des batteries, 92100 Châtillon", 16875432403ULL, 100438ULL );
    display(*pt_robert); ...
}
```

Les unions

Type contenant **exclusivement** un des types donnés dans l'union
On peut passer d'un des types à un des autres en accédant au champ correspondant

Syntaxe

```
union nom_type {  
    type_1 nom_champs1;  
    ...  
    type_n nom_champs_n;  
};
```



```
#include <iostream>  
union little_big_endian_detector {  
    unsigned lvalue;  
    unsigned char bytes[4];  
};  
int main() {  
    little_big_endian_detector detector;  detector.lvalue = 0x0000FFFF;  
    if ( detector.bytes[0] == 0xFF ) std::cout << "Little endian detected" <<  
std::endl;  
    else std::cout << "Big endian detected" << std::endl;  
    return EXIT_SUCCESS; }
```

Les énumérés

Ensemble de valeurs restreintes nommées énumérateurs de type dénombrable

Syntaxe

```
enum nom_type [: type] {  
    nom_1 = valeur1,  
    ...  
    nom_n = valeurs_n  
};
```

Type : uniquement C++ 11



```
enum base_color { red = 1, green = 2, blue = 4 };  
enum directions { x_dir = 0, y_dir, z_dir };  
enum orientation : unsigned char { direct = 0, undirect = 1 };
```

```
std::array<double, 3> point;  
point[x_dir] = 0., point[y_dir] = 1.; point[z_dir] = 0.;
```

Règle conseillée par les règles de bonne programmation isocpp :

toujours exprimer une valeurs constante par un énuméré ou une variable constante.

Exercices sur les structures et les unions

En utilisant une union, créer un programme affichant la représentation hexadécimale d'un nombre réel double précision en mémoire.

A partir d'un nuage de points calculés, construire la boîte alignée aux axes la plus petite possible qui englobe tout le nuage en complétant le programme `bouding_box.cpp` donné dans le répertoire `exercices/introduction`.

GESTION DES ERREURS EN C++

Catégories des erreurs, exceptions et
erreurs systèmes



Les exceptions

- Permet une gestion des erreurs
- A utiliser dans un cas rare ou exceptionnel
L'utilisation des exceptions fait partie de la conception de l'interface.
- Les exceptions sont extensibles (rajout facile de nouvelles exceptions);
- Permet un saut automatique de contexte;
- Délègue le traitement de l'erreur à l'appelant.

Principe d'utilisation

- **throw** lance un signal d'erreur (de type entier, string, autre...);
- **try** { ... **code protégé** ... }
 catch(type& e1) { ... **traitement erreur1** ... }
 catch(type& e2) { ... **traitement erreur2** ... }
 Mise en bloc de la zone à protéger.
- Les erreurs non rattrapées sont renvoyées plus haut dans la pile d'appel.

Exemple d'utilisation des exceptions

```
std::pair<double,double> find_root( double b, double c )
{
    double delta = b*b - 4*c;
    if ( delta < 0 ) throw std::string("Negative discriminant");
    double sqrt_delta = std::sqrt(delta);
    return {0.5*(-b+sqrt_delta), 0.5*(-b-sqrt_delta)};
}
```

Levée de l'exception

```
int main()
{
    double sol1, sol2;
    try {
        std::tie(sol1,sol2) = find_root(3,-1);
        std::cout << "OK, roots are : " << sol1
                  << " and " << sol2 << std::endl;
    }
    catch( std::string& msg ) {
        std::cerr << "Error finding roots : " << msg
                  << std::endl;
    }
    ...
}
```

Protection d'une zone de code

On rattrape l'erreur éventuelle

Exceptions prédéfinies

Dans **stdexcept** :

- Quelques exceptions prédéfinies, prévues pour être étendues
- **Les erreurs de logiques**
 - **logic_error** : Exception dûe à une erreur de logique
 - **domain_error** : Les valeurs des arguments sont en dehors du domaine de définition
 - **invalid_argument** : les arguments ne sont pas dans un état valide pour la fonction
 - **length_error** : La taille d'un conteneur n'est pas celle attendue
 - **out_of_range** : Indice non valide lors de lecture/écriture valeur dans un conteneur.
- **Les erreurs à l'exécution**
 - **runtime_error** : Erreurs lors de l'exécution du programme
 - **range_error** : Valeur obtenue en dehors de la représentation possible du type
 - **overflow_error** : Valeur trop grande pour le type de donnée
 - **underflow_error** : Valeur trop petite pour le type de donnée
 - **system_error** : Erreur système suite à l'interaction d'une librairie avec le système (inclure **system_error**)

De la bonne utilisation des exceptions

Gestion EAFP (Easier to Ask for Forgiveness than Permission)

```
if (std::abs(determinant(A))>1.E-14) {  
    inverse_system(A,b,x);  
}  
else {  
    project_solution(A, b, x);  
}
```

Calcul déterminant généralement inutile

```
try {  
    inverse_system(A,b,x);  
} catch(std::underflow_error&) {  
    project_solution(A,b,x);  
}
```

Code plus performant et plus clair !

Big Brother (Bigger protected block, better performance)

```
for ( int i = 0; i < nijk; ++i ) {  
    try {  
        comp_speed(rho[i],rho_u[i], u[i]);  
    } catch(std::underflow_error&) { ... }  
}
```

Gestion exception lente : fait à chaque pas de la boucle !!!

```
try {  
    for ( int i = 0; i < nijk; ++i )  
        comp_speed(rho[i],rho_u[i], u[i]);  
} catch(std::underflow_error&) { ... }
```

Temps gestion exception négligeable si boucle suffisamment longue.

Gestion des erreurs de programmation

- Aide au débogage de programme
- Arrête un programme si une condition n'est pas vérifiée
- Peut-être désactivée avec l'option **-NDEBUG**
- Permet de formaliser des prédicats en entrée et en sortie d'une fonction

```
#include <cassert>
...
double mean( const std::vector<double>& positiv_data )
{
    assert(positiv_data.size() > 0);
    double sum = 0.;
    for ( const auto& val : positiv_data ) {
        assert(val >= 0.);
        sum += val;
    }
    assert(sum > 0.);
    return sum/positiv_data.size();
}
```

Pré-condition

Post-condition

Exercices

- Reprendre la résolution d'un trinôme de second degré sur les réels en gérant le cas où le discriminant est nul afin de pouvoir basculer dans ce cas là dans le corps des complexes.
- Reprendre l'exercice sur l'orthonormalisation de Gram-Schmidt et gérez les erreurs que l'on peut éventuellement rencontrer sachant que l'utilisateur doit normalement donner une famille libre de vecteurs en entrée du programme.

Attention : Pour les deux exercices, il faudra d'abord veiller à mettre sous forme de fonction la résolution d'équation de second degré ainsi que l'orthonormalisation de Gram-Schmidt.



LES ENTRÉES-SORTIES EN C++



Registre
des entrées et sorties
de chiens ou de chats

Les entrées/sorties par flux

Plusieurs bibliothèques pour diverses fonctionnalités : `iostream`, `fstream`, `iomanip`,...

Ouverture des fichiers en lecture/écriture

- Utiliser le fichier d'entête `fstream`
- Pour ouvrir en lecture : `std::ifstream`
- Pour ouvrir en écriture : `std::ofstream`
- On peut rajouter des « modes » pour l'écriture :
 - ✓ Suppression de l'ancien fichier (par défaut)
 - ✓ Rajout à un fichier existant, sans modification possible des données déjà stockées
 - ✓ Rajout à un fichier existant avec modification possible des données déjà stockées

```
#include <fstream>
```

```
...
```

```
std::ifstream ifich("nom_fichier_à_lire");
```

```
std::ofstream ofich("nom_fichier_à_écraser");
```

```
std::ofstream ofich2("nom_fichier_à_augmenter", ios::app | ios::out);
```

```
std::ofstream ofich3("nom_fichier_modifier_augmenter", ios::ate | ios::out );
```

Ajouter sans
modification

Fermeture
fichier :

```
fich.close();
```

Tester si ouverture s'est
correctement déroulée :

```
if (not ifich) {  
    std::cerr << "Aie aie aie"  
    << std::endl;  
}
```

Ajouter avec
modification

Lecture/écriture dans un fichier

Lecture/écriture dans un fichier formaté

Ecriture simple :

```
int i = 3; double x = 3.14;  
std::string txt("# Dummy variables");  
ofich << i << " " << x << " " << txt;
```

Lecture simple :

```
int i; double x; std::string txt;  
ifich >> i >> x >> txt;
```

Contrôle du format pour un fichier sauvegardé (iomanip)

- std::setprecision(n) : Nombre de chiffres après la virgule à sauvegarder
- std::fixed, std::scientific, std::hexfloat, std::defaultfloat : divers formats d'affichage
- std::setw(n) : Affiche le prochain « objet » sur n caractères
- std::setfill(c) : remplit les « trous » par le caractère c

```
#include <iomanip>  
int main() {  
    std::cout << std::setprecision(14) << std::setw(30) << std::setfill('#')  
        << acos(-1) << std::endl;
```


Autres utilitaires pour la lecture formatée de fichiers

- Lire une ligne entière : **fich.getline(buffer, n[, delim])**
Lit au maximum n caractères. Fin de ligne définie par o.s sauf delim précisé
- Lire un caractère : **ifich.get(c)**
- Lire n caractères : **ifich.get(str, n)**
- Ignorer n caractères : **ifich.ignore(n)**
- Lire un caractère sans avancer dans le fichier : **c = ifich.peek()**
- Positionne le pointeur de lecture dans le fichier : **fich.seekg(offset,pos)**
- Indique la position du pointeur dans le fichier : **pos = fich.tellg()**

```
std::cout << "Enter name or number : " << std::endl;
std::cin >> std::ws; // Supprime les espaces en entête
int c = std::cin.peek();
if ( c == EOF ) return 1;
if ( std::isdigit(c) ) { ←
    int n;  std::cin >> n;
    std::cout << "Id entered : " << n << std::endl;
} else {
    std::string str; std::cin >> str;
    std::cout << "Name entered : " << str << std::endl;
}
```

Test si c est un chiffre (fichier entête ctype)

Lire/Ecrire dans un fichier binaire

Lecture binaire

```
ifich.read( char* buffer, std::size_t size );
```

écriture binaire

```
ifich.write( const char* buffer, std::size_t size );
```

```
std::ifstream infile ("test.txt",std::ifstream::binary);  
std::ofstream outfile ("new.txt",std::ofstream::binary);
```

Ouverture fichier binaire

// get size of file

```
infile.seekg (0,infile.end);
```

Va à la fin du fichier

```
long size = infile.tellg();
```

Retourne position = taille fichier

```
infile.seekg (0);
```

Retourne début fichier

// allocate memory for file content

```
char* buffer = new char[size];
```

// read content of infile

```
infile.read (buffer,size);
```

// write to outfile

```
outfile.write (buffer,size);
```

// release dynamically-allocated memory

```
delete[] buffer;
```

```
outfile.close();
```

```
infile.close();
```

Copie un fichier dans
une autre

Autres entrées sorties

Lecture/Ecriture dans une chaîne de caractère

- Permet de former dynamiquement des chaînes de caractères ou pouvoir lire des valeurs dans des chaînes de caractère
- S'utilise comme une entrée/sortie de fichier, de console, etc...
- Bibliothèque **sstream** à inclure

```
int n = 3; double x = 3.14, y;  
std::ostringstream foo;   
foo << "Facile d'ecrire " << n << " " << x << std::endl;  
std::string s = foo.str();  
  
std::string vals("3.14 334 2.57");  
std::istringstream fii(vals);  
fii >> x >> n >> y;  
std::cout << "x : " << x << ", y : " << y << ", n : "  
    << n << std::endl;
```

Pour former une chaîne de caractère

Récupération de la chaîne de caractère

Pour lire des valeurs dans une chaîne de caractère

Opérateur de flux

- Opérateur << et >> définis pour les types de base;
- Possibilité de les définir pour ses propres types

Exemple : opérateur de flux non défini pour les tableaux dynamiques

```
std::ostream& operator << ( std::ostream& out, const std::vector<double>& u ) {
    out << u.size() << "\t" << std::setprecision(14);
    for ( const auto& val : u ) out << val << " ";
    return out;
}

std::istream& operator >> ( std::istream& inp, std::vector<double>& u ) {
    std::size_t n; inp >> n;
    std::vector<double>(n).swap(u);
    for ( std::size_t i = 0; i < n; ++i ) inp >> u[i];
    return inp;
}

int main()
{
    std::vector<double> vect{1.,2.,3.,4.,6.,8.,-3.141517};
    std::cout << " vect : " << vect << std::endl;
    std::cin >> vect;
    std::cout << " vect : " << vect << std::endl;
```

Exercice sur les entrées sorties

Sortir dans un fichier un petit histogramme

Entrez des entiers : 3 4 5 7 1 12 8 4

```
+++ 3
++++ 4
+++++ 5
+++++++ 7
+ 1
+++++ 12
+++++++ 8
++++ 4
```

Indications :

- `std::getline(std::cin,txt);`
- `std::istringstream iss(txt);`
- `std::vector<std::string> integers(std::istream_iterator<std::string>{iss},
std::istream_iterator<std::string>());`
- `std::string(n,c)` : Crée une chaîne de caractères contenant n caractères c

#include <iterator>

Bonus : Fixer la largeur maximale à la plus grande valeur

`std::dynamic_pointer_cast<Derived>(base)`

Idem static, const et reinterpret cast (17 pour
reinterpret)