

Introduction à C++ 2011

Xavier JUVIGNY

ONERA

8 Juillet 2017

Plan du cours

1 Prérequis et finalité du cours

2 Introduction

Prérequis et finalité du cours

Prérequis

- Une bonne expérience en programmation
- Une bonne maîtrise du C

Finalité du cours

- Connaître les concepts fondamentaux de la programmation structurée, objet et fonctionnelle
- Maîtriser les concepts d'interface, d'encapsulation et de réutilisabilité;
- Être capable de concevoir des classes performantes et relativement complexes;
- Être capable de concevoir un programme C++ relativement complexe;
- Notion de programmation par template;
- Initiation à la Standard Template Library (STL).

le C++ en résumé

Historique

- **1969** : Unix DEC PDP-7, langage B issu de BCPL;
- **1972** : Portage d'Unix sur DEC PDP-11 : langage C;
- **1980** : C++, dérivé du C, inspiré de **Simula67** et **Algol68**;
- **1998** : Première normalisation du C++ par l'ISO
- **2011** : 2^e normalisation : refonte du langage
- **2014** : 3^e normalisation : corrections mineures
- **2017** : 4^e normalisation : politiques d'exécutions, ...;
- **2020** : 5^e normalisation en cours de proposition.

Paradigmes du C++

- Langage orienté objet, structuré, fonctionnel, universel.

Aperçu du langage

```
// Ceci est un commentaire sur une ligne
#include <iostream> // <--- inclusion module d'affichage
#include <string> // <--- module chaîne de caractères

/* Fonction disant bonjour à <nom>
   Input : <nom> le nom de la personne à qui
           le programme dit bonjour.
*/ // <--- Commentaire multiligne
void dit_bonjour( const std::string& nom )
{ // <--- début d'un bloc d'instruction
  std::cout << "Bonjour " << nom << "." << std::endl;
} // <--- fin d'un bloc d'instruction

// Programme principal. Retourne toujours un entier
int main (int nargs, const char* argv[]) { -- -->
  if (nargs == 1) return EXIT_FAILURE;
  dit_bonjour( argv[1] );// <--- appel de la fonction
  return EXIT_SUCCESS;
}
```

Type de retour de la fonction

Nom de la fonction

Liste arguments de la fonction

Arguments d'entrée pour le main

Syntaxe de base

- Toutes les instructions finissent par un ;
- Le début et la fin d'une fonction (dont le main) sont encadrées par {}
- Un exécutable doit toujours définir un `main` mais par une bibliothèque
- Une fonction peut ne rien retourner en déclarant son type de retour comme `void`

compilation du source

Comment créer l'exécutable ?

- 1 Se doter d'un compilateur C++ : g++, icpp, clang++, icl, ...compatible c++ 2014/2011
- 2 ou bien d'un ide intégré (visual c++, codeblock, etc.)
- 3 Ligne de commande standard : y

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

- 4 Remplacer `-std=c++14` par `-std=c++11` pour C++ 2011
- 5 Pour visualiser table des symboles (très difficile sinon sans l'option `-C`) :

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

- 6 Pour vérifier si on colle au standard à la compilation : `-pedantic`
- 7 Pour pouvoir déboguer, passer l'option `-g -O0`
- 8 Pour obtenir un code optimisé :
 - 9 Sur g++/clang++ : `-march=native -O3`
 - 9 Sur icc, `-xHost -O3`

Variables entières

Les booléens

Type ne prenant que deux valeurs : `true` (= 0) ou `false` (\neq 0)

```
// booléens
bool b1, b2, b3;
b1 = true; b2 = false;
b3 = b1 or b2; // b3 = true
b3 = b1 and b2; // b3 = false
b3 = b1 xor b2; // b3 = true ( ou exclusif )
b3 = (2>3); // b3 = false;
b3 = b1 and ((2>3) or (3-3==0));
b3 = not b1; // b3 = false
if (b2) { b3 = b1 or b2; } else { b3 = b1 and b2; }
```

Les entiers

- Beaucoup d'entiers : 8, 16, 32 ou 64 bits, signés ou non signés
- Entiers 8 bits

```
signed char ib = 3; // -128 <= ib <= 127
ib += 128; // ib vaut -125 !
```

```
unsigned char uib = 3; // 0 <= uib <= 255
uib -= 4; // uib vaut 255...
```

Ne jamais utiliser `char` pour entier 8 bits : signé ou non selon les os

Les entiers...

- Entiers 16 bits

```
short si = 68; // -32768 <= si <= 32767
si += 32700; // si vaut -32768
```

```
unsigned short usi = 36; // 0 <= usi <= 65535
usi += 65500; // usi vaut 0
```

- Entiers 32 bits

```
int i; // -2^31 <= i <= (2^31)-1
unsigned int ui; // 0 <= i <= (2^32)-1
```

- Entier 32/64 bits (selon os 32 ou 64 bits)

```
long li;
unsigned long uli;
```

- Entier 64 bits

```
long long lli;
unsigned long long ulli;
```

- Opérations usuelles : +, *, / (division entière), - (binaire ou unaire), % (modulo), - (décrémentation), ++ (incrémentation)
- Promotion automatique : `i + li` // <-- calcul en long

Réels et complexes

Les réels

- Trois types de réels possibles :
 - Simple précision (32 bits) : `float`
 - Double précision (64 bits) : `double`
 - Précision étendue (Au moins 64 bits) : `long double` (exemple : 80 bits si on utilise FPU d'Intel)
- Opérations arithmétiques usuelles : `+`, `-`, `*`, `/` (division réelle)
- Module `cmath` : fonctions mathématiques `cos`, `sin`, `tan`, `sqrt`, `exp`, `log`, `ln`, ...

```
double x,y; // Réel flottant 64 bits ( double précision )
long double lx; // Réel étendu
x= 3.14; y = 0.5; x = x + y; y += 2 * x * x;
y = std::cos(x); // <--- #include <cmath> en début de fichier
```

Les complexes

- Obligatoire inclure `# include <complex>` en entête de fichier.
- Divers complexes possibles, dépend de ce qu'on lui passe entre `<>`
- Opérations arithmétiques usuelles, partie réelle/imaginaire : `z.real()`; `z.imag()`;
- Fonctions mathématiques : `std::exp(z)`, `std::cos(z)`, ...;

```
std::complex<float> fz; // complexe simple précision
std::complex<double> z; // complexe double précision
std::complex<long double> lz; // complexe précision étendue
std::complex<int> iz; // complexe d'entiers
```


Chaînes de caractère

Déclarations

- Avec type de base : `const char* str;`
- Avec type protégé (inclure en entête de fichier `string`) : `std::string str;`

Opérations permises

- Concaténation chaîne de caractères : `str3 = str1 + str2;`
- Comparaison, insertion, recherche, etc...
- Convertir chaîne de caractères en entier, réel, etc. : `std::stoi`, `std::stod`, etc...
- Par défaut, codage ISO, mais utf8, utf16 ou utf32 possibles;
- Existe un type brute n'interprétant pas les caractères. Délimités par trois caractères + (et) + les trois caractères ;

```
std::string str1, str2, str3;
str1 = "tin"; str2 = "et"; str3 = "Pilou";
str1 += str1 + " " + str3;
size_t pos_pilou = str1.find(str3); str1[pos_pilou] = 'M';
str1[0] = std::toupper(str1[0]);
str1.insert(pos_pilou, str2+" "); std::cout << str1 << std::endl;
int i; double x;
str2 = "123"; str3 = "3.14";
i = std::stoi(str2); x = std::stod(str3);
std::cout << i << " : " << x << std::endl;
str1 = u8"chaînes avec accents en utf8";
str2 = R"RAW(C'est une chaîne
    multiligne qui marche bien)RAW";
str3 = u8R"RAW(Caractères utf8 raws !!!!)RAW";
```

Portée et visibilité des variables

Visibilité des variables

- Règles usuelles : visible dans son bloc et les sous-blocs
- Possibilité déclaration hors bloc : variable globale : à éviter si possible
- Deux variables ne peuvent porter le même nom dans un bloc
- Mais possible dans un bloc et un sous-bloc et peuvent être de types différents;
- Tant qu'une variable n'est pas déclarée dans un bloc, c'est la variable du bloc supérieur (ou au dessus) qui est visible;
- Une variable est détruite et son espace libéré à la sortie du bloc où elle a été déclarée;

```
int i = 0;

int main()
{
    std::cout << i << std::endl;
    int i = 1;
    std::cout << i << " != " << ::i << std::endl;
    {
        int i = 2;
        std::cout << i << " != " << ::i << " != " << std::endl;
        // Par contre, impossible d'atteindre le i du main...
    }
}
```

Les pointeurs

Types de pointeurs

Un pointeur est une valeur représentant une adresse de la mémoire du programme. Trois types de pointeurs existent en C++:

- **Le pointeur C** : uniquement un entier long représentant une adresse mémoire
- **Le pointeur unique** : un seul pointeur peut pointer sur une valeur donnée; La donnée est détruite quand plus aucun pointeur ne pointe dessus;
- **Le pointeur partagé** : plusieurs pointeurs peuvent partager la même donnée qui sera détruite lorsque plus aucun pointeur ne pointera sur la valeur.

Utilisation des pointeurs C

- Déclaration :

```
double x = 3.14, y = 1.54, z = 2.28;
int i = 3;
double array[3] = { 1., 2., 3. };
double *pt_x = &x;
double *pt_y = &y, *pt_z = &z;
double* *pt_pt_x = &pt_x; //Pointeur sur le pointeur de x
double *pt_arr = array; // Pointeur sur le 1er élément de array
```

- Accès à la valeur :

```
*pt_arr = 4; // On affecte la valeur 4 à l'élément pointé par pt_arr
pt_arr[2] = 1; // on affecte 1 à la valeur située 2 éléments plus loin
```

- Un pointeur particulier, le pointeur nul : `nullptr`

Arithmétique des pointeurs C

Incrémentation/décrémentation

Utile seulement pour les tableaux

- **Pré incrémentation** : Déplace le pointeur sur la valeur suivante et retourne le pointeur modifié; `++pt_arr;`
- **Pré décrémentation** : Déplace le pointeur sur la valeur précédente et retourne le pointeur modifié; `--pt_arr;`
- **Post incrémentation** : Déplace le pointeur sur la valeur suivante et retourne l'ancienne valeur du pointeur; `pt_arr++;`
- **Post décrémentation** : Déplace le pointeur sur la valeur précédente et retourne l'ancienne valeur du pointeur; `pt_arr--;`
- les pré incrémentation/décrémentation plus efficaces que les post...

Arithmétique de pointeur

- opération addition, soustraction permises :

```
pt_arr += 3; // pt_arr pointe trois valeurs après celle pointée
double* pt_arr2 = pt_arr - 2; // pt_arr2 pointe deux valeurs avant pt_arr
*(pt_arr2+1) = 3; // Equiv. à pt_arr2[1]
```

- Uniquement valable pour les pointeurs de type C;

Pointeurs uniques

Déclaration

- 1 Doit inclure `memory` en en-tête de fichier;
- 2 Permet de construire une valeur dynamiquement;
- 3 Un seul pointeur à la fois peut pointer sur cette valeur;
- 4 Si aucun pointeur ne pointe dessus, la valeur est détruite;
- 5 Pas d'arithmétique de pointeur pour ce type de pointeur;
- 6 Même interface que les pointeurs C;
- 7 Possibilité d'accéder au pointeur C correspondant.

```
std::unique_ptr<double> un_pt; // Pointe par défaut sur nullptr
std::unique_ptr<double> un_pt_v = std::make_unique<double>(4.56);
std::unique_ptr<double> un_pt_w = std::move(un_pt_v); // un_pt_w pointe sur 4.56, mais
                                                    // un_pt_v ne pointe plus sur rien.

double a = un_pt_w[0];
std::cout << *un_pt_w; // Même interface que pointeur standard mais pas d'inc/décrémentations
// A la fin du bloc, le dernier pointeur à pointer sur la valeur la détruit en mémoire.
```

Pointeurs partagés

Déclaration

- 1 Doit inclure `memory` en en-tête de fichier;
- 2 Permet de construire une valeur dynamiquement;
- 3 Plusieurs pointeurs peuvent pointer sur cette valeur;
- 4 Si aucun pointeur ne pointe dessus, la valeur est détruite;
- 5 Possibilité de connaître à chaque instant le nombre de pointeurs partageant cette valeur;
- 6 Pas d'arithmétique de pointeur pour ce type de pointeur;
- 7 Même interface que les pointeurs C;
- 8 Possibilité d'accéder au pointeur C correspondant.

```
std::shared_ptr<double> un_pt; // Pointe par défaut sur nullptr
std::shared_ptr<double> un_pt_v = std::make_shared<double>(4.56);
std::shared_ptr<double> un_pt_w = un_pt_v; // un_pt_w pointe sur 4.56
                                           // un_pt_v pointe sur 4.56

int ref_count = un_pt_w.use_count(); // Nombre de pointeurs pointant sur 4.56 ( ici 2 )
un_pt_v = std::make_shared<double>(3.14); // un_pt_v pointe sur 3.14
ref_count = un_pt_w.use_count(); // Cette fois ci, on retourne 1
un_pt_w = nullptr; // un_pt_w pointe sur nul, la valeur 4.56 est détruite car aucun pointeur
                    dessus
// La valeur 3.14 est détruite lorsque un_pt_v est détruit à la fin du bloc ( aucun pointeur
                    dessus )
```

Références

Définition d'une référence

- Renommage d'une variable;
- Variable partageant la même valeur qu'une autre variable (même espace mémoire).

Déclaration en C++

- Déclaration d'une référence à l'aide du symbole &;
- Doit être affectée obligatoirement à une autre variable à sa déclaration;
- Ou bien passée comme paramètre d'une fonction (voir plus loin dans le cours).

```
int i = 0;
int& ref_i = i;
int j = 1; // j vaut 0
ref_i = 3; // Affecte 3 à la valeur de i et de ref_i
j = 4; // j vaut 4 et i vaut 3
std::cout << "i = " << i << " = " << ref_i << " != " << j << std::endl;
```

Qualifieurs

Constantes

- Permet de définir une valeur non modifiable (exemple : pi);
- Si cette valeur est un scalaire de type entier ou réel, elle n'est pas stockée en mémoire mais remplacée dans le code;
- Seul un entier constant permet de définir un tableau de taille fixe (voir plus loin dans le cours);
- Permet de garantir qu'une valeur référencée ou pointée ne puisse être modifiée intentionnellement;
- L'opérateur `const_cast` permet d'enlever le qualifieur constant si besoin pour une valeur effectivement stockée en mémoire;
- L'opérateur `const_cast` appliqué à une variable définie à l'origine comme `const` a un comportement indéfini !;
- Cependant son utilisation traduit en général une mauvaise conception de l'interface.

```
const int A[] = { 1, 2, 3, 4 };
const int& i = A[0];
// i = 0 génère une erreur...
// A[0] = -1; génère une erreur
//const_cast<int&>(A[0]) = -1;// Comportement indéfini
const int n = 10;
double arr[n];
const double* pt_arr = arr;
const_cast<double&>(pt_arr[0]) = 3.14;
std::cout << arr[0] << std::endl;
//const_cast<int&>(n) = 15;// Compile mais gros bogue !
```

Volatile

Empêche pour une valeur certaines optimisations (la stocker dans le registre du processeur par exemple) dans le cas où cette variable peut être modifiée par un autre processus; (mot clef `volatile`)

Initialisation des variables

Alias de type

```
using dcomplex = std::complex<double>;
```

déclaration explicite

```
// Compatible ISO 98
int      i1 = 4, j1 = 2;
double   x1( 0.5 ), y1( 1.3 );
unsigned long u1( 9834541u );
std::size_t addr1 = 0xFFA2045C;
dcomplex z1( 0.5, 1.2 );
// Compatible uniquement 2011 ou supérieur :
int      i2{4}, j2{2};
double   x2{3.4}, y2{1.3};
unsigned long u2{9834541u};
std::size_t addr2{0xFFA2045C};
dcomplex z2{0.5, 1.2};
// Compatible 2014 ou supérieur :
unsigned long u3 = 9'834'541u; // Ecriture du nombre avec séparateurs
std::size_t addr3{0xFF'A2'04'5C}; // Idem pour la notation hexadécimale
// Nouvelle écriture binaire en C++ 14 :
std::size_t addr4 = 0b11111111'10100010'00000100'01011110; // Même adresse que addr3
double ex = 3'432.4123'6543;
// Indispensable pour utiliser la nouvelle notation pour les complexes
using namespace std::complex_literals;
dcomplex z3 = 0.5 + 1.2i;
```

Déclarations automatiques des variables

déclaration implicite

Déduction statique du type par le compilateur

```
auto i = 1;           // De la valeur à droite de =, on déduit que i est un int
auto x = 0.5 * i;      // Idem, ici on déduit que x est un double
auto fx = 0.5f * i;    // Et ici, fx est un float
auto& fy = fx;         // fy est une référence sur un float
auto const& fz = fx;    // et fz une référence constante
auto const* pt_x = &fx; // pointeur double constant sur pt_x
// auto z : Interdit car on ne peut pas déduire le type
i = 3.5;              // i prend la valeur 3, car toujours un int !
```

déclaration déductive

Déduit le type d'une variable à partir d'une expression

```
decltype( 1 + 3.5 )    xx; // xx est un double car 1+3.5 renvoie un double...
decltype((i1+x1)) rx = xx; // déduit type double qu'il met en référence
decltype(x1<0) b = x1<0; // Déduit le type boolean
decltype( std::norm( z1 ) ) dz; // dz est double car la fonction std::norm renvoie ici un double
// decltype permet d'avoir une déduction implicite pour les pointeurs
using ret_abs_t = decltype(std::abs(z1));
std::shared_ptr<ret_abs_t> pt_res = std::make_shared<ret_abs_t>(std::abs(z1));
```

TD 1 : prise en main

Problème

Écrire une petite calculatrice permettant d'effectuer des opérations élémentaires.

Aide

Chercher sur en.cppreference.com comment utiliser `std::stod`, `std::stol`, etc.

Bonus

Généraliser la calculatrice à une calculatrice polonaise inversée

```
• try {  
  ...  
  std::stod(...);  
  ...  
} catch(std::invalid_argument) {  
  // traitement opérateur  
}
```

• Regarder `std::stack<(type élément)>` sur en.cppreference.com

Accès par valeurs, pointeur et références

Accès par valeurs

On copie la valeur à droite du signe =

```
double a = 3; double b = a;  
a = 4;  
std::cout << "b = " << b << std::endl; // Affiche 3
```

Accès par pointeur

On copie l'adresse de la variable à droite du signe =

```
double a = 3; double* pt_b = &a;  
a = 4;  
std::cout << "*pt_b = " << *pt_b << std::endl; // Affiche 4
```

Accès par référence

On définit un alias de la variable à droite du signe =

```
double a = 3; double& b = a;  
a = 4;  
std::cout << "b = " << b << std::endl; // Affiche 4  
b = 2;  
std::cout << "a = " << a << std::endl; // Affiche 2
```

Conversion

Conversion explicite

Conversion sans vérification

```
double x = 1.5;
int i = int(x); // i = 1
const std::string s("Coucou");
const char* cs = s.c_str();
int* pt_i = (int*)&s; // Fonctionne mais...on ne peut rien en faire.
std::cout << *pt_i << std::endl;
pt_i = (int*)cs;
(*pt_i)++;
std::cout << s << std::endl; // Affiche Doudou sur Arm, ? sur intel
```

Conversion statique

Le compilateur vérifie que la conversion soit possible

```
double x = 1.5;
int i = static_cast<int>(x); // i = 1
const std::string s("Coucou");
const char* cs = s.c_str();
// const int * pt_i = static_cast<const int*>(&s); Erreur car ne sait pas comment convertir
const double A[] = {1.,2.,3.};
// double* pt_x = static_cast<double*>(A); Erreur car conversion de const à non const
const double* pt_x = static_cast<const double*>(A);
```

Structure

Structure

- 1 Syntaxe définition :

```
struct nom_nouveau_type {  
    type1 nom1;  
    type2 nom2;  
    ...  
    type_ret fonction1( args ) [const] { ... }  
    ... };
```

- Permet la définition de nouveau type contenant plusieurs types déjà définis;
- Accès champs à champs, possibilité de définir tous les champs en un coup;
- On peut aussi définir des fonctions associées aux données contenues dans la structure (on verra plus en détail dans le chapitre sur l'objet).

```
struct employe_onera {  
    std::string nom, prenom;  
    int id_badge, num_ldap;  
};  
employe_onera chambier { .nom = "Chambier", .prenom = "Robert", .id_badge=7, .num_ldap=1007 };  
std::cout << chambier.nom << std::endl;  
employe_onera *pt_e = &chambier; int id = pt_e->id_badge;
```

Exercice

Écrire un programme permettant pour un point donné en 2D de calculer si il appartient ou non à un triangle (utiliser les aires signées).

Union

Union

• Syntaxe :

```
union nom_nouveau_type {  
    type1 nom1;  
    type2 nom2;  
    ... };
```

- Permet de définir un type pouvant contenir exclusivement un des types listés;
- La taille d'une variable de type union est la plus grande des tailles des types donnés dans l'union.

```
union long_int_64 {  
    unsigned long val;  
    unsigned int  vals[2];  
};  
long_int_64 i;  
i.vals[0] = 10; i.vals[1] = 12;  
std::cout << i.val << std::endl;
```

Exercices

- Écrire un programme permettant d'écrire la représentation hexadécimale d'un réel double précision : Regarder documentation de `std::hex` sur internet

Les énumérés

Définition

- Permet une énumération de valeurs; Généralement représentés par un chiffre en C++;
- Énumération fixée à la compilation; Pas possible de rajouter de nouveaux éléments;
- Permet de restreindre à certaines valeurs une catégorie;
- Valeurs d'un énuméré non modifiables;

```
enum Couleur {
    Black = 0,
    Red   = 1,
    Blue  = 2,
    Violet = Red + Blue,
    Green = 4,
    Maroon = Red + Green,
    Yellow = Blue + Green,
    EndCouleur
};

std::array<std::string,EndCouleur> colourName = { "black", "red", "blue", "violet", "green",
                                                  "marron", "yellow" };

int main()
{
    std::cout << "One colour : " << colourName[Yellow] << std::endl;
```


Tableaux statiques

Déclarations et initialisations

bibliothèque : `array` : Défini des tableaux alloués à la compilation

```
# include <array>
std::array<unsigned,3> grid_indices;
std::array<double,5> arr{ {1.0035, 1.0,, 0., 0., 0.45} };
std::array<double,4> arr2 = {1.01,0.0,1.0};
std::array<int,2> dirs = {1,2,3}; // Erreur !
```

Accesseurs

Permet d'accéder aux données du tableau :

```
arr.size(); // Nombre d'éléments contenus dans le tableau
arr.data(); // Renvoie un pointeur sur le premier élément du tableau
arr.fill(val); // Rempli le tableau avec la valeur val
arr.swap(arr2); // Permute les valeurs de arr et arr2
```

Compareurs

Permet de comparer deux tableaux

```
if ( arr == arr2 ) {
    ...
} // Marche pour tous les compareurs ( ==, !=, <, >, <=, >= )
```

Tableaux dynamiques

Dynamique

bibliothèque vector : Des tableaux alloués à l'exécution et dont la taille peut être modifiée

```
# include <vector>
std::vector<double> vecFld; //Tableau vide encore non alloué
std::vector<int> indices(nijk); // Tableau de taille ni*nj*nk
std::vector<float> sol(nijk, 0.f); // Tableau taille nijk init. à 0
std::vector<std::vector<double>> mat; // Tableau de tableau...
std::vector<double> coef{ { 1., 1.5, 2., 2.5 } };
std::vector<std::string> token = { "parallel_for", "conv_flux" };
std::vector<double> varr(arr.data(), arr.data()+3);
```

Accesseurs

```
sol.size(); // Nombre d'elements contenus
mat.emplace_back({-1., 4., -1., 0.}); // Rajoute un élt en fin de tableau
sol.push_back(3.5f); // Rajoute un élément en fin de tableau
sol.pop_back(); // Enlève le dernier élt.
vecFld.reserve(5*nijk); // Alloue 5*nijk élt
vecFld.data(); // Pointeur sur l'adresse du 1er élt.
sol.resize(nijk/2); //Redimensionne le tableau
sol.capacity(); // Nombre d'elements reserves
sol.shrink_to_fit(); // Realloue le tableau à la bonne taille
std::vector<double>(123).swap(sol); // échange 2 tableaux
```

Accès directs aux éléments d'un tableau

Accès direct

- 1 Sans vérification d'indice :

```
// Pour gnu c++ :  
// -----  
// -D_GLIBCXX_DEBUG : Pour vérifier les indices  
// -D_GLIBCXX_DEBUG_PEDANTIC : Encore + de vérif.  
arr[7] = 13;
```

- 2 Avec vérification d'indice :

```
arr.at(13) = 257; // Génère une erreur de type std::out_of_range
```

- 3 Accès premier élément du tableau

```
arr.front() = 2.;
```

- 4 Accès dernier élément du tableau

```
arr.back() = 2.;
```

Accès séquentiels aux éléments d'un tableau

Accès séquentiel en avant

- 1 Par boucle, en mode C :

```
for (int i = 0; i < arr.size(); ++i ) arr[i] = 0.;
```

- 2 Avec itérateur (mode C++ 98) :

```
for ( auto it = arr.begin(); it != arr.end(); ++it ) *it = 0.;
```

- 3 Par boucle, en mode C++ 11 :

```
for ( double& val : arr ) val = 0.;
```

Accès séquentiel en arrière

- 1 Par boucle, en mode C :

```
for (int i = arr.size()-1; i >= 0; --i ) arr[i] = 0.;
```

- 2 Avec itérateur :

```
for ( auto it = arr.rbegin(); it != arr.rend(); ++it ) *it = 0.;
```

Autres conteneurs

Liste doublement chaînée

Bibliothèque : `#include <list>` :

```
std::list<double> lst1;  
std::list<int> lst2{ {1,3,5,7,11} };  
lst1.emplace_back(3.5);  
lst1.emplace_front(3.1);  
lst2.push_back(13); lst2.pop_front();  
for ( auto it = lst2.begin(); it != lst2.end(); ++it ) *it = -1;  
for ( auto& val : lst1 ) val = 3.1415;
```

Ensemble

Collection de valeurs uniques. Bibliothèque : `#include <set>`

```
std::set<int> indices{ {0, 3, 10, 13} };  
indices.insert(10); // 10 existe déjà, ne l'insère pas.
```

Dictionnaire

Bibliothèque : `#include <map>`. Existe aussi avec fonction de hashage (`unordered_map`).

```
std::map<std::string,int> id_personnes;  
id_personnes["philippe"] = 8905;  
id_personnes["Gérard"] = 9103;  
std::map<std::string,int> annee_perso { {"philippe", 1983},  
                                         {"gérard" , 1967} };
```

Les tuples

Les paires

- 1 Permet de définir une paire d'objet de type différents
- 2 Création à l'aide de `std::make_pair`
- 3 Accès au premier et second élément : `p.first` et `p.second`

```
auto p = std::make_pair(1, "Premier");  
std::cout << p.first << " et " << p.second << std::endl;  
auto p2 = std::make_pair(std::ref(p), "First");
```

Les tuples

- 1 Comme une paire mais avec un nombre quelconque d'objets
- 2 Création à partir de `std::make_tuple`
- 3 On peut récupérer des données d'un tuple à l'aide de `std::get` ou `std::tie`
- 4 Permet de dissocier pour une fonction les paramètres d'entrées (en argument) des paramètres de sortie (en retour avec un tuple si besoin).

```
auto trigo( double x ) { return std::make_tuple(std::cos(x),std::sin(x),std::tan(x)); }  
...  
double c,s;  
std::tie(c,s,std::ignore) = trigo(3.1415);  
auto tpl = std::make_tuple("Chat", 3.14, 5);  
std::cout << std::get<0>(tpl) << ", " << std::get<1>(tpl) << " et " << std::get<2>(tpl) << std::endl;
```

Manipulation des conteneurs

Générer l'ensemble des nombres impairs

Au moins deux solutions possibles

Statistique d'une chaîne de caractère

- Compter les minuscules, les majuscules, les chiffres, autres
- Afficher en fin de programme les caractères trouvés (en un seul exemplaire chacun)

Hint : regarder doc de `#include <cctype>`

Bonus : Gérer un carnet d'adresse

- fiche personne : nom, prénom, adresse, num. téléphone, num.sécurité sociale.
- Rajouter une personne;
- Rechercher une personne par son nom et prénom
- Afficher toutes les personnes de la base de donnée.

Déclaration des fonctions

Déclaration

● Façon C :

```
double sqr (double x ) {  
    return x*x;  
}
```

● Façon C++ 11 :

```
auto sqr (double x ) -> double {  
    return x*x;  
}
```

● Façon C++ 2014 :

```
auto sqr (double x ) {  
    return x*x;  
}
```


Modes de passage des arguments

Par valeurs

Copie de l'argument. Valeur de l'argument immuable.

```
void f( std::vector<double> a ) // <-- on recopie dans a
                                // le vecteur passé en argument
{ ... }
f(arr);
```

Par pointeurs

Passage explicite de l'adresse de l'argument. Valeur de l'argument modifiable.

```
void f( std::vector<double>* pt_a )
{ ... pt_a->resize(...); ... }
f(&arr);
```

Par référence

Référence sur l'argument. Valeur de l'argument modifiable

```
void f( std::vector<double>& a )
{ ... a.resize(...); ... }
f(arr);
```

Passage par référence

Passage par référence

Utiliser dans la signature d'une fonction pour

- Pouvoir modifier la variable passée en paramètre;
- Éviter de copier la variable passée en paramètre.

Inconvénients :

- Impossible de passer directement une valeur non stockée dans une variable;
- Variable passée en paramètre modifiable pour éviter une copie.

Passage par référence constante

Rajout du mot clef `const` dans la signature

- Éviter une copie sans rendre modifiable la variable passée en paramètre;
- Permet de passer des valeurs.

Exemples passage par référence

```
using vecteur = std::vector<double>;

double normalize_inplace ( vecteur& u )
{ ... }

void transpose_inplace ( const vecteur& tr, vecteur& u )
{ ... }

int main ()
{
    vecteur u(3);
    ...
    normalize_inplace ( u ); // Normalise u
    transpose_inplace ( vecteur{ {1., 0., -1.} }, u );
    ...
    // Erreur : on ne peut pas passer une valeur
    normalize_inplace ( vecteur{ {1., 1., 1.} } );
}
```

Retour d'une fonction

Retour par valeur

```
std::vector<double> f (...)
```

Deux modes de retour induits :

- 1 **Par copie** : pour une variable non locale ou un scalaire → pour un tableau, on le recopie élément par élément dans un nouveau tableau.
- 2 **Par déplacement** ; pour une variable locale ou si on spécifie à l'aide de l'instruction `std::move` → pour un tableau, on crée un nouveau tableau qui va "voler" le pointeur du tableau retourné.

Retour par référence

```
std::vector<double>& f (...)
```

Retourne un "alias" sur une variable qui doit être

- 1 Un paramètre de la fonction;
- 2 une variable globale.

Ne jamais retourner une variable locale par référence !

Surcharge des fonctions et paramètres par défaut

Surcharge fonctions

Possibilité de redéfinir même fonction avec signatures différentes.

```
int pow_n( int x, int n ) {  
    return (n==0 ? 1 : n&1 ? x*pow_n(x*x,(n-1)/2) : pow_n(x*x,n/2));  
}  
float pow_n( float x, int n ) {  
    return (n==0 ? 1 : n&1 ? x*pow_n(x*x,(n-1)/2) : pow_n(x*x,n/2));  
}  
double pow_n( double x, int n ) {  
    return (n==0 ? 1 : n&1 ? x*pow_n(x*x,(n-1)/2) : pow_n(x*x,n/2));  
}  
int main() { ...  
    int iy = pow_n( 13, 5 );  
    int y = pow_n( 11.5, 7 );  
}
```

Paramètres par défaut

Paramètres optionnels. Toujours derniers paramètres. Uniquement définis à la déclaration.

```
double daxpy( int n,const double* x,double* y,int incx=1, int incy=1);  
...  
double daxpy( int n, const double* x, double* y, int incx, int incy ) {  
    ...  
}  
int main() { ...  
    d = daxpy( arr.size(),arr.data(),arr2.data() ); // incx=1, incy=1  
    d2 = daxpy( arr.size(),arr.data(),arr2.data(),2 ); // incx=2,incy=1  
    d3 = daxpy( arr.size(),arr.data(),arr2.data(),2,2 ); // incx=2, incy=2  
}
```

Surcharge des opérateurs

Redéfinition d'un opérateur

- Possibilité de redéfinir des symboles
- Pour opérateurs binaires, premier argument correspond à la valeur à gauche de l'opérateur.

```
std::vector<double> operator +( const std::vector<double>& u,  
                                const std::vector<double>& v )  
{  
    ...  
}  
...  
auto w = u + v;
```

Opérateur de flux

Pour lire/écrire un nouveau type sur fichier/console/etc.

```
std::ostream& operator << ( std::ostream& out,  
                             const std::vector<double>& u ) {  
    ...  
    return out;  
}  
std::istream& operator >> ( std::istream& inp, std::vector<double>& u ) {  
    ...  
    return inp;  
}
```

Liste des opérateurs (non exhaustif)

Opérateurs arithmétiques				
+	binaire	<code>t operator + (const t&, const t&)</code>		Addition
-	unaire	<code>t operator - (const t&)</code>		Opposé
-	binaire	<code>t operator - (const t&, const t&)</code>		Soustraction
*	binaire	<code>t operator * (const t&, const t&)</code>		Multiplication
/	binaire	<code>t operator / (const t&, const t&)</code>		Division
%	binaire	<code>t operator % (const t&, const t&)</code>		Modulo
Opérateurs d'affectation et inplace				
=	binaire	<code>t& operator = (t&, const t&)</code>		Copie
+=	binaire	<code>t& operator += (t&, const t&)</code>		Addition inplace
-=	binaire	<code>t& operator -= (t&, const t&)</code>		Soustraction inplace
*=	binaire	<code>t& operator *= (t&, const t&)</code>		Multiplication inplace
/=	binaire	<code>t& operator /= (t&, const t&)</code>		Division inplace
%=	binaire	<code>t& operator %= (t&, const t&)</code>		Modulo inplace
Opérateurs de comparaison				
==	binaire	<code>bool operator ==(const t&, const t&)</code>		Comparaison égalité
>	binaire	<code>bool operator >(const t&, const t&)</code>		Supérieur à
<	binaire	<code>bool operator <(const t&, const t&)</code>		Inférieur à
<=	binaire	<code>bool operator <=(const t&, const t&)</code>		Inférieur ou égal à
>=	binaire	<code>bool operator >=(const t&, const t&)</code>		Supérieur ou égal à
!=	binaire	<code>bool operator !=(const t&, const t&)</code>		Différent de
Opérateurs d'accès				
[]	binaire	<code>(const) t& operator [] ((const) t&, const Ind&)</code>		Accesseur mono indice
*	unaire	<code>(const) t& operator * ((const) pt_t&)</code>		Opérateur de dérèrencement
->	unaire	<code>(const) t& operator ->((const) pt_t&)</code>		Opérateur d'accès pointeur
Pré/Post incrément				
++	unaire	<code>t operator ++ (t&)</code>		Pré incrémentation
++	unaire	<code>t operator ++ (const t&, int)</code>		Post incrémentation
-	unaire	<code>t operator -- (t&)</code>		Pré décréméntation
-	unaire	<code>t operator -- (const t&, int)</code>		Post décréméntation
Autres				
()	n-aire	<code>Out operator () (arg1, arg2, ..., argn)</code>		Évaluateur/Accesseur

Les exceptions

Utilité

- Erreurs
- Traitement d'un cas rare où exceptionnel;

Les exceptions font partie de la conception de l'interface

Propriétés

- Extensible
- Saut automatique de contexte
- Délègue le traitement de l'erreur à l'appelant

Principe

- `throw` : lancer un signal d'erreur (entier, string, autre)
- `try { ... code protégé ... } catch(type1& e1) ... traitement erreur 1 ... catch(type2& e2) ... traitement erreur 2 ...` : Mise en bloc de la zone à protéger. les erreurs non rattrapées sont renvoyer plus haut dans la pile d'appel

Les exceptions - 2

La levée de l'exception

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

Protection d'une zone de code

```
try {  
    auto sols = find_root( 3, -1. );  
} catch(std::string& msg)  
{  
    std::cerr << "Erreur en cherchant les racines : " << msg  
               << std::endl;  
}
```

Les exceptions - 3

Exceptions prédéfinies

dans `stdexcept`

● Logic errors

- `logic_error` : Logic error exception
- `domain_error` : Domain error exception
- `invalid_argument` : Invalid argument exception
- `length_error` : Length error exception
- `out_of_range` : Out-of-range exception

● Runtime errors

- `runtime_error` : Runtime error exception
- `range_error` : Range error exception
- `overflow_error` : Overflow error exception
- `underflow_error` : Underflow error exception

Les exceptions - 4

Gestion EAFP

Easier to Ask for Forgiveness than Permission

```
if (determinant(A) != 0) {  
    inverse_system(A,b,x);  
} else {  
    project_solution(A,b,x);  
}
```

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

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&){ ... }  
}
```

```
try {  
    for ( int i = 0; i < nijk; ++i ){  
        comp_speed(rho[i],rho_u[i],u[i]);  
    }  
}
```

Les entrées/sorties (fstream)

Ouverture/création fichiers

- Création d'un flux en lecture/écriture :
 - `std::ifstream fich(nom_fich)` : Ouverture en lecture d'un fichier;
 - `std::ofstream fich(nom_fich)` : Ouverture en écriture d'un fichier (écrase l'ancien);
 - `std::ofstream fich(nom_fich,mode)` : Ouverture en écriture d'un fichier selon un ou plusieurs modes définis (exemple : `ios::out|ios::app`) :
 - `ios::out` : Ouverture en mode écriture (défaut)
 - `ios::trunc` : Tronque (efface) l'ancien fichier si il existait déjà et remplace par un nouveau;
 - `ios::app` : Rajoute à la suite du fichier si il existait déjà. Impossible d'écraser ce qui existait déjà même en se déplaçant en "début" de fichier;
 - `ios::ate` : Rajoute à la suite du fichier si il existait déjà. On peut écraser les données précédentes en se déplaçant en début de fichier par exemple.
- On peut tester si l'ouverture s'est correctement déroulé : `if (fich) ...`

Écrire/Lire dans un fichier formaté

● Écrire

```
int i = 5; double x = 3.14; std::string chaine("strange data...");  
fich << i << x << chaine;
```

● Lire

```
int i; double x; std::string chaine;  
fich >> i >> x >> chaine;
```

Les entrées/Sorties (suite)

Changer le format des nombres sauvegardés (iomanip)

- `std::setprecision(n)` : Écrit n chiffres après la virgule;
- `std::fixed`, `std::scientific`, `std::hexfloat`, `std::defaultfloat` : divers formats...
- `std::setw(n)` : Affiche le prochain "objet" sur n caractères.
- `std::setfill(c)` : Remplit les "trous" avec le caractère c

Autre utilitaire de lecture formatée

- `fich.getline(buffer, n, delim)` : Lit ligne entière (n caractères max). Fin ligne définie par o.s, sauf si `delim` précisé.
- `fich.get(c)` : Lit le caractère c dans le fichier
- `fich.ignore(n)` : Lit sans stocker n caractères du fichier;
- `fich.peek()` : Lit le caractère courant sans avancer dans le fichier;
- `fich.seekg(offset, pos)`/`fich.tellg()` : Met/Donne la position dans le fichier

Écrire/lire dans un fichier binaire

- Écrire : `fich.write(buffer, size)`; où `char* buffer`;;
- Lire : `fich.read(buffer, size)`; où `char* buffer`..

Fermer un fichier

- `fich.close()`

Autres entrées/sorties

Lecture/Écriture dans une chaîne de caractères (sstream)

- Permet de former dynamiquement des chaînes de caractères; Même interface que pour un fichier + quelques autres opérations

```
# include <sstream> ...
int main() {
    int n = 3, double x = 5, y;
    std::ostringstream foo; foo << "Facile d'écrire " << n << " " << x << std::endl;
    std::string s = foo.str(); std::string vals("3.14 334 4.14");
    std::istringstream fii(vals);
    fii >> x >> n >> y;
    return 0; }
```

Exercice (difficile): Gestion d'une bibliographie

- Écrire une structure pouvant contenir une bibliographie primaire (Titre, auteur, année, éditeur, type);

```
Bibliography bib;
Bibliography::item book1{title="Vie des douzes césars", author="Sueton",
                          year=121, editor="?", type="Biography"};
add(bib, book1);
```

- Un opérateur permettant d'afficher/écrire un livre sur console/fichier;
- Écrire une fonction qui sauvegarde la bibliographie, une fonction qui la lit;
- Une fonction cherchant l'ouvrage le plus récent/Ancien de la bibliographie.

```
auto& old_book = bib_oldest(bib); auto& new_book = bib_newest(bib);
```

Les espaces de nommage

Conflits de noms

- Deux entités portant le même nom. Exemple : `Vecteur` pour un vecteur soit géométrique soit algébrique;
- Ou deux fonctions portant le même nom et la même signature.
- En C++, on peut résoudre cela en utilisant des espaces de nommage :
 - Soit en utilisant l'espace de nommage dans nos propres headers :

```
namespace Geometry {  
    struct Vecteur { std::array<double,3> coefs; };  
}  
namespace Algebra {  
    struct Vecteur { std::vector<double> coefs; };  
}
```

- Soit en encapsulant l'inclusion dans l'espace de nommage pour des headers externes :

```
namespace Geometry {  
# include <geometry/vecteur.hpp>  
}  
namespace Algebra {  
# include <algebra/vecteur.hpp>  
}
```

```
Geometry::Vecteur x;  
Algebra::Vecteur u;
```

Utilisation des espaces de nommage

Possibilité

- 1 Ne pas préciser l'espace de nommage;

```
namespace Algebra {  
    struct Vecteur...  
}  
{  
    Algebra::Vecteur u;  
}  
{ // Tout ce qui est dans Algebra sera visible dans l'espace global  
    using namespace Algebra;  
    Vecteur u;  
}  
{ // Seul Algebra::Vecteur est visible dans l'espace de nommage global.  
    using Algebra::Vecteur;  
    Vecteur u;  
}
```

- 2 Renommer un espace de nommage

```
namespace Numeric {  
    namespace Integration {  
        double quadrature( double x0, double x1, double(*f)(double x));  
    }  
}  
{  
    using NumInteg = Numeric::Integration;  
    ...  
    x = NumInteg::quadrature(0, 2, std::sqrt);  
    ...  
}
```


Les expressions constantes

Définition

- Valeurs évaluées au moment de la compilation;
- Peuvent être des valeurs de types définis par l'utilisateur;
- On peut utiliser des expressions relativement complexes.

```
const int constexpr N = 100;
const int constexpr M = N + 21;
const int constexpr O = ( M%2 == 0 ? M/2 : 3*N+1);
```

Fonctions en expressions constantes

- Des fonctions peuvent être définies comme expressions constantes pour évaluer ces valeurs;
- Elles peuvent être utilisées comme fonction normale ou pour évaluer une expression constante;
- Si uniquement pour évaluer une expression constante, pas compilée et n'existera pas dans l'application finale;
- Des structures/classes peuvent être évaluées en expressions constantes ainsi que leurs méthodes.

```
// On utilise la méthode de Newton pour trouver la racine carrée de x:
//  $x_{n+1} = x_n - f(x_n)/f'(x_n) = x_n - (x_n^2 - x)/(2 \cdot x_n)$ 
double constexpr sqrt_NewtonRaphson( double x, double curr, double prev )
{ return curr == prev ? curr : sqrt_NewtonRaphson(x, 0.5*(curr + x/curr), curr ); }
// On gère quand même les cas où la racine n'existe pas !
double constexpr cst_sqrt(double x)
{ return x >= 0 && x < std::numeric_limits<double>::infinity() ?
    sqrt_NewtonRaphson(x,x,0) : std::numeric_limits<double>::quiet_NaN(); }
```

```
const double constexpr sqrt2 = cst_sqrt(2);
```

Les expressions constantes...

Règles à suivre pour les fonctions expressions constantes

- Ne pas utiliser des fonctions de la librairie standard (sauf extension comme dans gnu);
- Ne pas être virtuelles (voir le chapitre sur l'objet pour les fonctions virtuelles);
- Doit retourner un type littérale : type scalaire, tableau de type littéraux, constructeur constexpr et destructeur trivial (voir chapitre sur l'objet), void, agrégat, union, ne pas être volatile;
- Pas de boucles en C++11 (mais possible en C++14);
- Pas de goto, de asm, de définitions de variables non littérales, de variables statiques, de blocs `try...catch...`, de variables statiques.

```
// Calcul de la racine d'un vecteur 3D en expression constante
double constexpr normL2( double x, double y, double z ) { return cst_sqrt(x*x+y*y+z*z); }
/* Le calcul de pi peut se faire à l'aide d'une triple suite. Cette formule double
   en décimal le nombre de chiffre exact après la virgule à chaque itération.
   Cette formule est inspirée de celle de brent - salamin ( 1976 ) :
   A_0 = 1;           A_{n+1} = (A_n + B_n)/2.
   B_0 = sqrt(1/2);   B_{n+1} = sqrt(A_n * B_n)
   C_0 = 1/4;         C_{n+1} = C_n - 2^{-n}*((A_n-B_n)/2)^2
   et on calcul pi_n comme pi_n = (A_n+B_n)^2/(4*C_n)
*/

double constexpr brent_salamin( int n, double an, double bn, double cn,
                                double cur, double prev ) {
    return cur == prev ? cur : brent_salamin(n+1, 0.5*(an+bn), cst_sqrt(an*bn),
                                              cn - (1<n)*0.25*(an-bn)*(an-bn), 0.25*(an+bn)*(an+bn)/cn, cur);
}
```

```
const double constexpr sqr3 = normL2(1.,1.,1.);
const double constexpr pi = brent_salamin(0,1.,cst_sqrt(0.5),0.25,3.,0.);
double x = 3, y = 5, z = 8;
double nrmp = normL2(x,y,z);
```

A vous de jouer !

Gestion de matériaux

- Matériaux identifiés par leurs coefficient de Young ν , température de fusion T et leur résistance électrique R ;
- Sauver/Lire des matériaux dans une base de donnée qu'on pourra charger entièrement en mémoire;
- Interroger la base pour connaître les caractéristiques d'un matériau.

Exemple de test

```
MDB_Materiaux db;
ifstream ifich("materiaux.db");
ifich >> db;
ifich.close();
std::cout << db["acier"].young << std::endl;
std::cout << db["chene"] << std::endl;
Materiau coton{ .young = 0.1, .T = 100, .R = 50 };
db["coton"] = coton;
ofstream ofich("materiaux.db", "w");
ofich << db;
ofich.close();
```