RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

Introduction
Introduction

Xavier JUVIGNY, AKOU, DAAA, ONERA
xavier.juvigny@onera.fr

Initiation au C++
- 4 Novembre 2023 -

[1]ONERA,[2]DAAA

Table of contents

Overview

Historique

- **1969** : Première version d'UNIX écrit en assembleur sur DEC PDP-7
- **1970** : Tentative de porter UNIX sur autre machine avec langage B : trop lent et dépendant machine
- **1971** : Création du langage C par D. Richtie et portage d'UNIX en langage C
- **1980** : Création du C++ par Bjarne Stroustrup
- **1983** : Standardisation ANSI du langage C
- **1998** : Première norme ISO du C++
- **2011** : Seconde norme ISO du C++ : beaucoup de nouvelle caractéristiques
- **2014, 2017, 2020, 2023,...** : Evolution de la norme tous les trois ans depuis 2011

Caractéristiques du langage C++

- **Langage compilé** : Le langage C++ nécessite un compilateur qui traduit un fichier texte en langage machine directement interprétable par le processeur ;
- **Performant** : Aujourd'hui, le C++ est aussi, voir plus performance que le langage C ;
- **Multi-paradigme** : Différentes méthodologies de programmation peuvent être appliquées en C++, au sein d'un même code :
  - *La programmation structurée* : On décompose un problème en structures et fonctions ;
  - *La programmation orientée objet* : On se concentre sur la gestion des données, à savoir comment gérer ces données (protocole) et la manière dont elles sont gérées (interface)
  - *La programmation fonctionnelle* : On décompose le problème en un ensemble de fonction (au sens mathémtiques du terme) qu'on manipule (en les composant, etc.) afin de mettre en œuvre la fonctionnalité visée
- **Bibliothèque de base très riche en fonctionnalité**.

Remarque : Ne pas se limiter à un seul paradigme lorsqu'on programme en C++ !

Les différentes étapes de compilation en C++

- **La précompilation** : On remplace les macros par les textes correspondant et les "includes" par le texte contenu dans le fichier à inclure ;
- **Première phase compilation** : Une partie du code source est compilée afin de servir à la génération de données utiles à la seconde phase de compilation ;
- **Seconde phase compilation** : On transforme le reste du code en pseudo-assembleur en utilisant les données issues de la première phase ;
- **Troisième phase compilation** : Transforme le code pseudo-assembleur dans le langage machine du processeur visé ;
- **Edition des liens** : Assemble plusieurs fichiers "objets" contenant du langage machine pour former un exécutable ou une bibliothèque binaire.

Remarque : On peut se demander quelle est l'utilité de la première phase de compilation, propre au C++. En fait, il existe en C++ des instructions qui permettent d'évaluer certaines données (qui peuvent être des structures) à la compilation et non à l'exécution.

Choix de la norme C++

Chaque norme du C++ apporte son lot de nouveauté au langage (nouvelles fonctionnalités dans la bibliothèque standard, nouvelles caractéristique du langage, etc.).
Aujourd'hui, par défaut, les compilateurs prennent la norme C++ 17 comme norme de référence, mais il possible de demander une norme plus récente (mais généralement à laquelle il peut manquer une ou deux fonctionnalités), ou bien une norme plus ancienne pour assurer le portage pour des compilateurs plus anciens.
Néanmoins, il faut avoir conscience que les normes successives tendent à une simplification du langage, le rendent moins verbeux et moins dépendant de l'OS sur lequel on compile.
Dans cette formation, on précisera quand nécessaire à partir de quelle version du langage une fonctionnalité ou caractéristique donnée est apparue.
Lors de la compilation d'un fichier, si on veut contraindre ou étendre le compilateur à une version de la norme donnée, il faut utiliser l'option `-std=C++xx` où `xx` est la norme du C++ choisie.
Exemple : `-std=c++20` pour pouvoir utiliser des fonctionnalités du C++ 20.

Overview

La fonction main

En C et C++, une fonction particulière sert de point d'entrée pour les exécutable : la fonction `main` :

```
// Première forme
int main()
{
    ...
    return EXIT_SUCCESS;
}
```

```
// Seconde forme
int main( int nargs, char* argc[] )
{
    ...
    return EXIT_SUCCESS;
}
```

La fonction `main` retourne toujours en entier donnant l'état du programme à sa sortie : soit un succès soit un échec (`EXIT_FAILURE`).
La **seconde forme** prends en argument un entier donnant le nombre d'arguments passés à l'exécutable, ainsi qu'un tableau contenant les arguments passés sous forme de chaînes de caractère.

Attention : Le première argument passé est le nom de l'exécutable lui-même !

Les commentaires en C++

Il existe deux formes de commentaire :

- Sur plusieurs lignes, commençant par le marqueur /* et finissant par le marqueur */ ;
- Sur une seule ligne, commençant par le marqueur // et finissant par le retour à la ligne

```
// Exemple d'un commentaire sur une seule ligne
/* Exemple d'un commentaire
   sur plusieurs lignes.
*/
```

Attention : Les commentaires multilignes peuvent s'avèrer problématique à l'usage si on les imbrique par inadvertance :

```
/*  Imaginons qu'on veut commenter une zone du programme en utilisant le commentaire sur plusieurs lignes.
    C'est bien pratique
    /* Malheureusement, dans la zone qu'on a commenté, il existait déjà un commentaire
       multiligne... */
    On pourrait penser que nous sommes encore dans un commentaire.
    Quelle erreur !
*/
```

Les blocs d'instruction

- Les blocs d'instruction sont délimités par les symboles { et } ;
- Une variable déclarée dans un bloc d'instruction n'est visible que dans ce bloc d'instruction ;
- Une variable déclarée hors de tout bloc d'instruction est une **variable globale** ;
- Les fonctions, les boucles et les branchements conditionnels sont délimités par un bloc d'instruction.
- On peut créer un bloc d'instruction dans un bloc d'instruction (même si il n'est pas associé à une fonction, boucle, etc.)

Exemple :

```
int main()
{// Début de la fonction main

    {// Début d'un bloc d'instruction dans la fonction main

    }// Fin du bloc d'instruction défini dans la fonction main

}// Fin de la fonction main.
```

Les espaces de nommage

Région déclarative fournissant une portée aux identificateurs (noms de type, fonctions, variables, etc.) à l'"intérieur.

- Permet d'organiser le code en groupes logiques ;
- Permet d'éviter les conflits de nom

Exemple :

```
namespace géométrie { // On définit un espace de nommage nommé géométrie
    struct Vecteur { // On définit un nouveau type vecteur géométrique
    ...
    };
}
namespace algébre { // On définit un espace de nommage nommé algébre
    struct vecteur { // On définit un nouveau type vecteur algébrique
    ...
    }
}
géométrie::Vecteur u; // On déclare un vecteur géométrique
algébre::Vecteur   b; // On déclare un vecteur algébrique
```

Remarque : Toutes les composantes de la librairie standard sont dans l'espace de nommage `std`

Utilisation des espaces de nommage

Il est possible d'éviter dans le cas de toujours précéder un identificateur par l'espace de nommage où il a été déclaré :

- Pour un identificateur spécifique :

```
using géométrie::Vecteur;
Vecteur u; // C'est un vecteur géométrique
algèbre::Vecteur v;// Ici, on a spécifié, donc un vecteur algébrique
```

- Pour l'ensemble des identificateurs d'un epsace de nommage :

```
using namespace géométrie;
Vecteur u; // Un vecteur géométrique donc
```

Espaces de nommage imbriqués

Les espaces de nommage peuvent être imbriqués
Les identificateurs dans un espace de nommage imbriqué à accès aux membres de l'espace parent sans qualifié l'espace de nommage parent.
Par contre, l'espace de nommage parent doit utiliser l'espace de nommage de l'espace imbriqué sauf si ce dernier est "inline".

```
namespace A {
    namespace B {
        const int degré = 2;
    }
    const int ordre = B::degré + 1;
}
```

Il est possible à partir de C++ 17 de définir directement plusieurs espace de nommage imbriqués :

```
namespace monalisa::géométrie
{
    struct Vecteur { ... };
}
monalisa::géométrie::Vecteur u;
```

Espace de nommage "inline" (à partir du C++ 11)

Contrairement à un espace de nommage imbriqué ordinaire, les membres d'un espace de nommage "inline" sont traités en tant que membres de l'espace de nommage parent.

```
namespace opérateur_intégral
{
    namespace ancienne_version {
        int calcul(...);
        ...
    }
    inline namespace nouvelle_version {
        int calcul(...);
        ...
    }
    void evaluate() {
        ...
        calcul( ... );
        ...
    }
}
```

Utilisation des bibliothèques

Bibliothèque standard très riche en fonctionnalités :

- Gestion de la mémoire
- Chaînes de caractères
- Conteneurs : tableaux, listes, dictionnaires, ...
- Fonctions mathématiques avancées
- Générations avancées de nombres aléatoires
- Algorithmes de tri, de recherche,etc. séquentiels et parallèles
- Gestion fichiers, chronomètre, multithreading, etc.

De manière générale, consulter le site cppreference
Richesse du langage et de sa bibliothèque ⇒ impossible de tout maîtriser. Bjarne Stroutrup, créateur du C++, prétend aujourd'hui ne maîtriser que 70% du langage !
Pour utiliser une des composantes d'une bibliothèque (standard ou système) :

```
#include <fichier à inclure>
```

Pour utiliser une composante d'une bibliothèque appartenant à son projet :

```
#include "fichier à inclure"
```

Pour les composantes des bibliothèques non standards, les fichiers à inclure possèdent conventionnellement les extensions "hpp" ou "h".

iothèque standard, les fichiers à inclure ne possèdent pas d'extensions.

Déclaration des variables

Syntaxes :

```
Type nomVariable; // Pour une seule variable
Type nomVar1, nomVar2, ..., nomVarN; // Pour plusieurs variables de même type
```

Les variables peuvent être déclarées à tout endroit du code.
Il est possible d'initialiser la variable à sa déclaration :

```
Type nomVariable{valeur};// Initialise une variable à sa déclaration
Type nomVar1{valeur1}, nomVar2{valeur2},...;// Initalise plusieurs variables à leurs déclarations
```

Note : Il est possible de remplacer les accolades par des parenthèses ;
Parmi les types de bases proposés par le C++ sont :

- Chaînes de caractère "basiques" : `char *var{"valeur"};`
- Entiers (défaut) : `int var{valeur};`
- Réels 64 bits : `double var{valeur};`
- Booléens : `bool var{valeur};`

Note : Il est préférable d'éviter autant que possible des variables de type `char *`, le C++ proposant par ailleurs une bibliothèque gérant de façon plus simple les chaînes de caractère.

Affichage sur le terminal

Par défaut, le C++ est incapable d'afficher quoi que ce soit sur le terminal. Il faut utiliser une composante de la bibliothèque standard : `iostream`
Deux sorties sur le terminal sont possibles :

- La sortie standard : `std::cout`
- La sortie erreur : `std::cerr`

Pour afficher sur ces deux sorties, on utilise l'"opérateur de flux" << qu'on peut chaîner pour afficher plusieurs valeurs.
Pour faire un retour chariot, on utilise la valeur définie dans la bibliothèque : `std::endl`
Par exemple, en supposant qu'on veut afficher un message, la valeur d'un entier `i` et retourner sur une nouvelle ligne :

```
std::cout << "La valeur de i est " << i << std::endl;
```

Notre premier programme :

```
#include <iostream>
int main() {
    std::cout << "Hello world !" << std::endl;
    return EXIT_SUCCESS;
}
```

Les noms des variables

- Toute variable doit commencer par :
  - une lettre latine minuscule ou majuscule
  - un underscore
  - un caractère d'un sous-ensemble de l'UTF8 (lettre autre que latine)
- Les autres caractères composant le nom de la variable peuvent être :
  - Un chiffre
  - Une lettre latine minuscule ou majuscule
  - Un underscore
  - Un caractère d'un sous-ensemble de l'UTF8 (lettres autre que latines ou chiffres en indices ou exposant)

Exemples de noms de variables valides

uneVariable, _nom, π, n^2, un_nombre, ...

Exemples de noms de variables non valides

1nombre, une variable, un$, ...

Attention : En C++, les noms de variables sont sensibles à la casse !

Les caractères

Un caractère représente une "lettre" ou un symbole. En C++, le type caractère est `char`.
On initialise un caractère ASCII en le mettant entre deux simple quote. Par exemple :

```
char c = '#';
```

Remarque : On peut également initialiser une variable avec le symbole =
Il est par contre impossible tel quel d'initialiser un simple ou double quote. Il faut utiliser le symbole
pour éviter au C++ d'éviter d'interpréter le simple ou double quote (ce qui sera vrai aussi pour les chaînes de caractère)
Il existe plusieurs symboles spéciaux :

- `'\n'` : Retour à la ligne suivante
- `'\t'` : Aller à la prochaine tabulation
- `'\r'` : Retour en début de ligne

Le type `char` étant codé sur un seul octet, il n'est pas possible d'initialiser un caractère C++ avec un caractère codé en UTF8
Il est possible d'initialiser un caractère directement avec le code ASCII correspondant :

```
char c = 97;// c contient le caractère 'a'
c = 98; // Maintenant c contient le caractère 'b'
```

Remarque : On change la valeur d'une variable en C++ à l'aide du symbole =

Les chaînes de caractère

Nous avons vu qu'il existait déjà `char *` pour déclarer une chaîne de caractère mais qu'il valait mieux éviter d'utiliser ce type.
En C++, il existe également une composante de la bibliothèque C++ : `string` qui permet de déclarer une chaîne de caractère plus facile et plus sûre à manier.
On déclare alors une chaîne de caractère avec le type `std::string`.

Initialisation d'une chaîne de caractère

- Déclaration d'une chaîne vide

```
std::string phrase;
```

- Déclaration d'une chaîne de caractère simple :

```
std::string phrase{"Une simple ligne"};
```

- Déclaration d'une chaîne de caractère litérale. Pour ces chaînes, le C++ n'interprète pas les symboles rencontrés dans la chaîne de caracère. Exemple :

```
std::string doc = R"DOC(
Un exemple de documentation  où  le
"retour chariot" est possible ainsi
que l'utilisation  des  'quotes' et
"double quotes"
 )DOC"
```

Services associés à une chaîne de caractère

Plusieurs services sont associés aux chaînes de caractère de type `std::string` :

- Nombre de caractères d'une chaîne `s` : `s.size();`
- Copier la chaîne `s1` dans une chaîne `s2` : `s1 = s2;`
- Lire le $i^{\text{ème}}$ caractère de `s` : `char c = s[i];`
- Modifier le $i^{\text{ème}}$ caractère de `s` : `s[i] = 't';`
- Insérer un caracèture ou une chaîne de caractère à la $i^{\text{ème}}$ position : `s.insert(i," toto ");`
- Concaténer deux chaînes `s1` et `s2` dans `s` : `s = s1 + s2;`
- Échanger deux chaînes de caractères `s1` et `s2` : `s1.swap(s2);`
- Trouver la première position d'une sous chaine à partir de la position `i` : `s.find("tintin", 3);`
- Transformer une valeur en chaîne de caractère : `s = std::to_string(3.14);`
- ...

Note : Comme tout conteneur (tableaux, chaîne de caractère, etc.), les indices commencent à 0 !

Définition directe d'une std ::string (C++ 14)

- Par défaut, les chaînes de caractères `"Une chaîne..."` sont de type `char *` ;
- Initialisation d'une std ::string avec une telle chaîne non optimale (copie de la chaîne de caractère) ;
- En utilisant l'espace de nommage `std::string_literals` on peut définir une std ::string directement en postfixant un `s` à la dernière double quote.

Exemple :

```
#include <string>
...
std::string une_chaine = " directement initialisee !"s;
```

Ces chaînes offrent tous les services d'une `std::string` normale :

```
std::cout << "une longue phrase contenant avant les deux points :"s.size() << std::endl;
```

Les entiers de base

Plusieurs entiers possibles de base en C++ :

- Entiers signés :

Type	Intervalle	Remarque
`signed char`	$[-128; 127]$	
`short`	$[-32768; 32767]$	
`int`	$[-32768; 32767]$ ou $[-2^{31}; 2^{31}-1]$	Dépend du système ou processeur
`long`	$[-2^{31}; 2^{31}-1]$ ou $[-2^{63}; 2^{63}-1]$	Dépend du système ou processeur
`long long`	$[-2^{63}; 2^{63}-1]$	

- Entiers non signés

Type	Intervalle	Remarque
`unsigned char`	$[0; 255]$	
`unsigned short`	$[0; 65535]$	
`unsigned`	$[0; 65535]$ ou $[0; 2^{32}-1]$	Dépend du système ou processeur
`unsigned long`	$[0; 2^{32}-1]$ ou $[0; 2^{64}-1]$	Dépend du système ou processeur
`unsigned long long`	$[0; 2^{64}-1]$	

Attention : Il possible d'utiliser le type `char` comme entier, mais il n'est pas spécifié dans la norme si il est signé ou non signé !

Contrôler la taille des entiers !

Le C++ est livré avec une composante de la bibliothèque standard permettant de contrôler la taille des entiers : `cstdint` On peut alors déclarer des entiers signés avec la syntaxe suivante : `std::intX_t` où `X` est le nombre de bits contenus dans l'entier (8, 16, 32 ou 64) Ainsi :

```
std::int8_t byte; // Entier sur 8bits signé
std::int32_t quad; // Entier sur 32 bits signé
```

Idem avec les entiers non signés avec la syntaxe `std::uintX_t`. Ainsi :

```
std::uint16_t uword; // Entier non signé sur 16 bits
std::uint64_t dqword; // Entier non signé sur 64 bits
```

Il existe également un entier non signé représentant l'étendu mémoire que peut gérer le processeur pour lequel est compilé le code : `std::size_t` qui est un entier non signé 64 bits sur les architectures et OS modernes, et 32 bits sur les processeurs ou OS un peu anciens.

Opérations sur les entiers

Les opérations possibles sur les entiers sont :

- Les opérations arithmétiques usuelles : `+, -, *, /` (ici la division entière)
- les opérateurs inplace : `+=, -=, *=, /=` : `a -= 4;`
- L'opération modulo `%` : `a = b % 4;`
- La pré incrémentation (rajoute un et retourne la nouvelle valeur) : `++i`
- La pré décrémentation (enlève un et retourne la nouvelle valeur) : `--i;`
- La post incrémentation (rajoute un et retourne l'ancienne valeur) : `i++`
- La post décrémentation (enlève un et retourne l'ancienne valeur) : `i--;`
- Transformer une chaîne de caractère en entier : `std::stoi(s), std::stol(s), std::stoll(s) ...`

Additionneur de deux entiers

```
#include <iostream>
int main( int nargs, char *argc[]) {
    int x = std::stoi(argc[1]), y = std::stoi(argc[2]);
    std::cout << x << " + " << y << " = " << x+y << std::endl;
    return EXIT_SUCCESS;
}
```

Formatage des entiers en sortie

- Utilisation de `iomanip`
- `std::setw` réserve un nombre de caractère à afficher
- `std::setfill` remplit l'espace non utilisé par un caractère.

```
std::int32_t value1 = -32;
std::int32_t value2 =  3 ;
std::cout << "value1 = " << value1 << std::endl;
std::cout << "et value2 = " << value2 << std::endl;
std::cout << "123456789ABCDEF" << std::endl;
std::cout << std::setw(15) << "value1 = " << std::setw(4) << value1 << std::endl;
std::cout << std::setw(15) << "et value2 = " << std::setw(4) << std::setfill('0') << value2 << std::endl;
```

Ce qui affiche

```
value1 = -32
et value2 = 3
123456789ABCDEF
      value1 =  -32
   et value2 = 0003
```

Nombres à virgule flottante

Il existe trois types de nombres à virgule flottante :

- `float` : Réel simple précision (32 bits)
- `double` : Réel double précision (64 bits)
- `long double` : Réel quadruple précision (128 bits en principe)

Opérations définies sur les nombres à virgule flottante :

- Opérateurs arithmétiques usuels : `+, -, *, /`
- Opérateur arithmétiques inplace : `+=, -=, *=, /=`

La composant `cmath` de la bibliothèque standard fournit nombre de fonctions mathématiques :

- Fonctions trigonométriques usuelles :
  `std::sin,std::cos, std::tan, std::asin, std::acos, std::atan, std::atan2`
- Fonctions hyperboliques usuelles : `std::sinh, std::cosh, std::tanh, std::asinh,std::acosh, std::atanh`
- Fonctions exponentielles et logarithmiques : `std::exp, std::expm1, std::log, std::log10, std::pow`
- racines, fma et modulo généralisé : `std::sqrt, std::cbrt, std::fma, std::fmod`
- fonctions mathématiques spécialisée : polynôme de Laguerre, de Legendre, fonction beta, intégrales elliptiques de première et seconde espèce, fonctions de bessel, etc. depuis le C++ 17

Nombres à virgule flottante spécialisés

Depuis C++11, trois valeurs spéciales définies dans `limits` :

- `quiet_NaN` : Not a Number sans mettre le processeur en état d'erreur. Sa comparaison d'égalité avec toute autre valeur dont lui-même renvoit toujours faux
- `signaling_NaN` : Not a Number en mettant le processeur dans un état d'erreur.
- `infinity` : Représente l'infini. Toujours supérieur à n'importe quel nombre réel.

```cpp
#include <limits>
#include <iostream>
int main() {
    double x = std::numeric_limits<double>::quiet_NaN();
    std::cout << std::boolalpha <<"Nan==NaN? "<< x==std::numeric_limits<double>::quiet_NaN() <<std::endl;
    std::cout << "x n'est pas un nombre ? " << std::isnan(x) << std::endl;
    float finf = std::numeric_limits<float>::infinity(), fsup = std::numeric_limits<float>::max();
    std::cout << fx << " < ∞ ? :" << (fx < finf) << std::endl;
    return EXIT_SUCCESS; }
```

```
Nan==NaN? : false
x n'est pas un nombre ? true
3.40282e+38 < ∞ ? true
```

Les constantes prédéfinies (C++ 20)

De nombreuses constantes usuelles sont définies dans la composante numbers de la bibliothèque standard.
Ces constantes sont sous deux formes :

- Prédinies pour les réels double précision : `std::numbers::pi`, `std::numbers::e`, ...
- Génériques pour d'autres types de réels : `std::numbers::pi_v<float>`, `std::numbers::e_v<float>`, ...

Exemple :

```
long double pi_lf = std::numbers::pi_v<long double>;
std::cout << std::setprecision((std::numeric_limits<long double>::digits10+1))
          << "pi = " << pi_lf << std::endl;
double invpi = std::numbers::inv_pi;
double e     = std::numbers::e;
double phi   = std::numbers::phi;
```

Les nombres complexes

Les nombres complexes ne sont pas natifs. Il faut include la composante `complex` de la librairie standard.
Il est possible alors de définir des nombres complexes dont le type des cœfficients est choisi par le programmeur. Par exemple :

```
std::complex<double> z1; // Complex double précision
std::complex<float>  z2; // Complex simple précision
std::complex<long>  iz1; // Nombre de Gauss: complexe contenant des entiers
```

Il existe deux manières d'initialiser un nombre complexe en C++ :

- Initialisation classique : `std::complex z{0.,1.};`
- Initialisation "naturelle". Il faut préalablement après avoir inclu `complex` utiliser un espace de nommage : `using namespace std::complex_literals`. On peut alors initialiser des complexes à coefficients réels comme suit :

```
std::complex fz = 3.f + 2.if; // Complex simple précision
std::complex dz = 3.  + 2.i ; // Complex double précision
std::complex lz = 3.L + 4.iL; // Complex long double précision
```

Remarque : Il n'est pas nécessaire depuis C++ 17 de préciser le type des coefficients lors de l'initialisation si les valeurs passées sont de même type.

Opérations sur les complexes

- Les opérations arithmétiques
- Les opérations arithmétiques inplace.
- L'opérateur d'égalité et de différence (== et !=)
- Obtenir la partie réelle `z.real()` ou imaginaire `z.imag()`
- Le module du complexe : `std::abs(z)`
- L'argument du complexe : `std::arg(z)`
- Le carré du module : `std::norm(z)`
- Le conjugué : `std::conj(z)`
- Initialise un complexe à partir de son module et argument : `z = std::polar(r,theta);`
- Fonction exponentielle : `std::exp(z)`
- Fonction logarithme : `std::log(z)`
- Fonction puissance : `std::pow(z1,z2)`
- Racinne carrée : `std::sqrt(z)`
- Les fonctions trigonométriques : `std::sin,std::cos,std::tan`
- Les fonctions trigonométriques inverses : `std::asin,std::acos,std::atan`
- Les fonctions hyperboliques : `std::sinh,std::cosh,std::tanh`
- Les fonctions hyperboliques inverses : `std::asinh, std::acosh, std::atanh`

Renommage de type

Il est possible de renommer des types lorsque ces derniers sont verbeux à l'aide de la commande `using`

Exemple :

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

Autre avantage de renomme des types est de pouvoir basculer facilement entre plusieurs représentation mémoire des données :

```
using conteneur = std::list<double>;
...
conteneur tasks;
...
tasks.push(...);
...
```

```
using conteneur = std::vector<double>;
...
conteneur tasks;
...
tasks.push(...);
...
```

Déclaration implicite du type (C++ 11 et supérieur)

On peut remarquer lors de la déclaration d'une variable accompagnée de son initialisation, que le type peut être implicitement déclaré deux fois. Par exemple :

```
std::complex z{1.,2.};
std::complex z2 = z*z; // z*z est un complex double, z2 ne peut que l'être aussi
```

On peut éviter dans ce cas de préciser le type qu'on initialise et laisser le compilateur déduire le type à l'aide du mot clef `auto` :

```
std::complex z{1.,2.};
auto z2 = z*z; // z*z est un complex double, z2 ne peut que l'être aussi
```

Nous verrons que ce mot clef est très utile et permet au c++ d'être beaucoup moins verbeux. Par exemple, en C++ 98 on écrivait :

```
for (std::vector<double>::const_iterator iter=u.begin(); iter != u.end(); ++iter)
{    ...
}
```

En C++ 11, on pourrait écrire cette boucle comme :

```
for (auto iter=u.begin(); iter != u.end(); ++iter)
{    ...
}
```

valeurs constantes et expressions constantes

Il est possible de définir une variable représentant une valeur constante, c'est à dire qu'on ne peut modifier. Il suffit pour cela après (ou avant) avoir défini le type de rajouter le qualificatif `const` :

```
int const nbEssais = 4;
nbEssais = 5; // Cette ligne provoque une erreur de compilation !
```

Sauf si une valeur constante est une simple valeur comme l'exemple ci-dessus, sa valeur sera évaluée à l'exécution :

```
double const pis3 = 1.0471975511965976;
double const cos_pis3 = std::cos(pis3);// cos_pis3 sera évaluée à l'exécution.
```

Il est néanmoins possible depuis C++ 11 de demander au compilateur à évaluer une valeur durant la phase de compilation. On utilise pour cela le mot clef `constexpr` :

```
constexpr double a00 = 1., a10 = 0.5, a01 = 0.75, a11 = 1.3;
constexpr double x = 2., y = -1.5;
constexpr auto valeur = a00 + a10*x + a01*y + a11 * x * y;
```

Si une partie de l'expression n'est pas évaluable en constexpr, l'évaluation se fera durant l'exécution.

Les fonctions mathématiques ne sont utilisables en constexpr que depuis C++ 23 (g++ le supporte aujourd'hui).

Branches conditionnelles

La syntaxe est simple :

```
if (cond) <bloc d'instructions>
else <bloc d'instruction>
```

Noter les parenthèses autour de la condition. Si condition est composée de plusieurs conditions articulées avec les opérateurs logiques, chaque condition doit être mis entre parenthèse et l'ensemble des conditions également.
Pour les opérateurs logiques, on dispose de

- `&&` ou `and` pour l'opérateur logique ET,
- `||` ou `or` pour l'opérateur OU
- `!` ou `not` pour l'opérateur NON.

Par exemple, pour tester si un entier `i` est impair et divisible par trois et le diviser par trois dans ce cas, sinon retrancher un à l'entier :

```
if ( (i%2 != 0) and (i%3 == 0) )
    i = i/3;
else
    i = i - 1;
```

Remarque : Il est inutile d'utiliser les accolades si le bloc d'instruction ne contient qu'une seule instruction.

Sélecteur de cas

Possible de chaîner les if de la manière suivante :

```
if (cond1) { ... }
else if (cond2) { ... }
else ...
```

Si beaucoup de cas à traiter selon la valeur d'une variable, on utilise `switch` et `case`. Par exemple, dans la gestion d'un menu, si on a une valeur entière `imenu` donnant la sélection de l'utilisateur :

```
switch(imenu) {
case 1: // Si imenu vaut 1
    liste d'instructions
    break;
case 2: // Si imenu vaut 2
    liste d'instructions...
    break;
...
default:// Autres valeurs non prévues
    std::cout << "Choix invalide !" << std::endl;
}
```

Sans mettre de `break` à la fin d'un `case`, le C++ continue avec les instructions se trouvant dans le cas suivant (fallthrough).

Les boucles de type while

Il existe deux types de boucles :

- Les boucles de type "Tant que (cond) faire ..." :

```
while (cond) <bloc d'instructions>
```

Exemple

```
int n=7;
while (n!=1) {
    if (n%2==0) n = n/2;
    else n = 3*n+1;
}
```

- Les boucles de type "Faire ... tant que (cond)" :

```
do <bloc d'instructions> while (cond);
```

Exemple

```
int imenu = 0;
do {
    std::cout << "Votre choix : ";
    std::cin.clear();
    std::cin >> imenu;
} while ( (imenu < 1) or (imenu > 5) );
```

Il est bien sûr possible de mettre pour condition la valeur `true` pour obtenir une boucle infinie.

Boucle for

La boucle `for` très souple. Prend deux formes, la boucle continuant tant que la condition est vraie :

- Première forme : les variables itérées sont déclarées avant la boucle :

```
for (init vars; cond; modif.)  <bloc d'instruction>
```

- Seconde forme : les variables itérées sont déclarées et initialisées dans le for :

```
for (décl. et init vars; cond; modif.)    <bloc d'instruction>
```

  Sous cette forme, les variables itérées ne sont visibles quand dans le bloc d'instruction de la boucle

Exemple 1 : Recherche du premier bit non nul d'un entier n à partir du bit le plus faible

```
int firstRightBit; for (firstRightBit = 0; (n%2==0) and (n>0); n = n/2, firstRightBit++);
```

Remarque : le bloc d'instruction peut ne contenir aucune instruction !

Exemple 2 : Calcul la puissance p d'un entier n

```
int pow_n = 1; for (int pow = 0; pow < p; ++pow ) pow_n *= n;
```

Boucle for(suite)

Au sein du bloc d'instruction :

- Il est possible d'arrêter une itération et continuer à la suivante avec l'instruction `continue`
- On peut également arrêter les itérations à l'aide de l'instruction `break`

Exemple 3 : Tester si un nombre entier n est premier (vrai) ou non (faux)

```
bool estPremier = n%2==0 ? false : n%3==0 ? false : true;
for (int i=1; (6*i-1 < std::sqrt(n)) && estPremier; ++i)
{
    if (n%(6*i-1) == 0) {
        estPremier = false;
        break;
    }
    if (n%(6*i+1) == 0) estPremier = false;
}
```

Remarque 1 : On a utilisé ici l'opérateur ternaire `cond ? éval si vrai : éval si faux;`

Remarque 2 : On verra une troisième forme pour la boucle for avec les conteneurs...

Exercices

- Résoudre l'équation quadratique $a.x^2 + b.x + c = 0$ avec a,b et $c \in \mathbb{R}$. a, b et c seront passés en argument à l'exécutable et la résolution se fera dans $\mathbb{R}$.
- Calculer la longueur de vol, c'est à dire le nombre d'itérations nécessaire pour retomber sur le cycle $1 \rightarrow 4 \rightarrow 2 \rightarrow 1$ avec la suite de Syracuse : $u_{n+1} = \begin{cases} u_n/2 \text{ si } u_n \text{ est pair} \\ 3 * u_n + 1 \text{ si } u_n \text{ est impair} \end{cases}$ u_0 sera donné en argument à l'exécutable
- Calculer et afficher les n premiers termes de la série

$$S_n = 4. \sum_{k=0}^{n} \frac{(-1)^k}{2k+1} \left(\frac{4}{5^{2k+1}} - \frac{1}{239^{2k+1}} \right)$$

n étant donné en argument à l'exécutable

Les références

Variable faisant **référence** à la valeur d'une autre variable : variable référente précédée par le symbole &.

Exemple

```
int i  = 2;
int &j = i; // j fait rérence à i et à sa valeur (ici 2)
i = 3; // La valeur de i est maintenant 3, et donc j fait référence à la valeur de i, donc 3
j = 4; // j faisant référence à i, la valeur de i est maintenant égale à quatre
```

Possibilité de déclarer une référence constante, c'est à dire que la variable référente ne peut modifier la valeur référée

Exemple

```
int i = 2;
int const &j = i;
i = 3; // j faisant référence à la valeur de i, affichera la valeur 3
j = 2; // Erreur de compilation : j ne peut modifier la valeur référée
```

Nous allons voir que les références sont un moyen très pratique de passer des arguments par adresse et non par valeur en C++

Déclaration des fonctions

Fonctions peuvent être déclarées avant d'être mise en œuvre en général dans un fichier *header* (extension `.h` ou `.hpp`).
La déclaration d'une fonction se fait avec la syntaxe suivante :

```
<type de la valeur de retour> nom_de_la_fonction( type nom_arg1, type nom_arg2, ... );
```

Possible de ne passer aucun argument à la fonction (une parenthèse ouvrante suivie d'une parenthèse fermante).
Si la fonction ne renvoie aucune valeur (équivalent procédure en fortran), la fonction renvoie le type `void` comme valeur de retour.

Exemples

```
void affiche_menu();
double solveLinearEquation( double a, double b );//Résoud ax+b=0
```

Plusieurs fonctions peuvent avoir le même nom si arguments différents en nombre ou en type (surcharge fonction) :

Exemples

```
float solveLinearEquation( float a, float b );//Résoud ax+b=0 en simple précision
double solveLinearEquation( double a, double b );//Résoud ax+b=0 en double précision
```

Définition des fonctions

Définir fonction dans fichier de mise en œuvre (extension cpp) avec la syntaxe suivante :

```
<type retour> nom_fonction( type1 param1, type2 param2, ... )
{   ...
    // Avec des returns si la fonction renvoit quelque chose.
}
```

Exemple

```
float solveLinearEquation( float a, float b ) {  return -b/a; }
```

Depuis C++ 11, il existe une seconde forme de syntaxe pour définir une fonction (forme fonctionnelle) :

```
auto nom_fonction( type1 param1, type2 param2, ...) -> <type de retour>
{ ... }
```

Exemple

```
auto solveLinearEquation( float a, float b ) -> float {  return -b/a; }
```

Passage par valeur/Passage par référence

Par défaut, les arguments passés à une fonction sont passés par valeur : on copie la valeur des variables passées en argument.

```
void inc(int n) { n += 1; }
...
int x = 3;
inc(x); // x vaut toujours trois à la sortie de la fonction !
```

Pour passer un argument par adresse, il faut le passer par référence :

```
void inc(int &n) { n += 1; }
...
int x = 3;
inc(x); // x vaut quatre à la sortie de la fonction !
```

Pour des valeurs volumineux en mémoire (une matrice par exemple), il est préférable de passer l'argument correspondant par référence. Si de plus, on veut s'assurer que l'argument ne puisse pas être modifié, on peut passer une référence constante :

```
auto solve(MatricePleine const& A, Vecteur const& b ) -> Vecteur;
```

L'avantage de la référence constante est de plus de pouvoir passer directement une valeur en argument :

```
auto y = solve(A, Vecteur{A.dimension(), 1.} );
```

Paramètres par défaut

- Possible de donner des valeurs par défaut à des paramètres de la fonction ;
- Ces paramètres doivent être passés en dernier dans la fonction ;
- On peut ne pas passer ces arguments si les valeurs par défaut conviennent ;
- Si on passe un argument ayant une valeur par défaut, il faut également passer les arguments le précédent, même si ils ont des valeurs par défaut.

Exemple :

```
std::complex<double> monopole( double kWave, double radius, double minRadius = 0.01, double phase = 0. )
{
    double minRad = std::max(radius, minRadius);
    return std::polar(1./minRad, kWave * minRad + phase );
}

auto p1 = monopole( 1.1, 0.5 ); // minRadius = 0.01, phase = 0.
auto p2 = monopole( 1.1, 0.01, 0.02); // minRadius = 0.02, phase = 0.
auto p3 = monopole( 1.1, 0.05, 0.1, std::numbers::pi/4. );// minRadius = 0.1, phase = pi/4
```

Surcharge des fonctions

Plusieurs fonctions peuvent possèder le même nom si et seulement si la signature des fonctions sont différentes, c'est à dire :

- Le nombre de paramètre diffère **OU**
- Le type des paramètres diffèrent

Attention : Le nombre et le type des données retournées par une fonction ne font pas partie de la signature de la fonction !

Exemple

```
float max( float x, float y ) {
  return x >= y ? x : y;
}

float max( float x, float y, float z ) {
  if (x < y) return y >= z ? y : z;
  return x >= z ? x : z;
}

double max( double x, double y ) {
  return x >= y ? x : y;
}
```

Surcharge des opérateurs

Opérateur

Symbole effectuant une opération : `+ - * / += -= *= /= = % ++ -- & && | || &= |= ^ < <= == >= > != , [] () ~ ->`

Note : Opérateurs ++ et -- peuvent être définis pour pré-incrémentation et post-incrémentation. Rajoute d'un entier inutilisé en paramètre pour définir post-incrémentation. Exemple :

```
Type operator ++ ( Type& t ) { ... } // Pré-incrémentation
Type operator ++ ( Type& t, int ) { ... } // Post-incrémentation
```

Possible redéfinir opérateurs pour nouveaux types. Exemple, supposons type `Point` défini :

```
Point operator + ( Point const& p1, Point const& p2 ) { ... }
bool operator == ( Point const& p1, Point const& p2 ) { ... }
Point operator * ( double scal, Point const& p ) { ... }
Point& operator *= ( Point& p, double scal) { ... }
Point& operator = ( Point& p1, Point const& p2 ) { ...; return p1; }
int main() {
  Point p1, p2;
  p1 = p1 + p2;// Utilise operateur + et operateur =
  p1 *= 2; // Utilise opérateur *=
```

Les assertions

Les assertions sont utiles lors de la phase de développement d'un code.
Elles vérifient qu'un test est vrai. Si celui-ci est faux, le programme s'arrête en affichant le fichier et la ligne où l'assertion fausse a eu lieu.
Les assertions sont très utiles pour :

- Vérifier si les paramètres d'entrées appartiennent au domaine de définition de la fonction appelée (Précondition) ;
- Vérifier si les valeurs de retour sont conformes à ce que doit retourner la fonction (Postcondition).

On utilise pour cela la composante `cassert` de la bibliothèque standard (std).

Exemple :

```
// Calcul la norme d'un vecteur 3D (x,y,z) et normalise le vecteur
double normalize( double &x, double &y, double &z )
{
    double sqrNorm = x*x + y*y + z*z;
    assert(sqrNorm > 1.E-14);
    double norm = std::sqrt(sqrNorm);
    x /= norm; y /= norm; z /= norm;
    assert(norm > 0);
    return norm;
}
```

Contrat d'une interface et mise en œuvre d'un algorithme

- **Contrat** : caractérisation de l'interface
  - Qu'est ce que l'algorithme est capable de produire ;
  - Domaine de définition de l'algorithme
  - Valeurs possibles à la sortie
- **Exemple** : *Racine carrée d'un réel*
  - En entrée : un réel positif ou nul
  - En sortie : un réel positif ou nul
- **Précondition** : Quelles conditions doivent vérifier les valeurs connues en entrée de l'algorithme ?
- **Postcondition** : Quelles conditions doivent vérifier les valeurs modifiées ou retournées en sortie de l'algorithme ?

Utilisation des assertions

- Utilisation des `assert` de la composante `cassert` de la bibliothèque standard ;
- Attention : `assert` vient du langage C et ne possède pas d'espace de nommage `std::` !
- On peut ne pas vérifier les assertions (par exemple pour un exécutable en production) à l'aide de l'option de compilation `-DNDEBUG`
- Exemple pour la racine carrée :

```
#include <cassert>
double sqrt( double x )
{
    assert(x >= 0); // Précondition
    double sq = ...; // Calcul de la racine carrée avec votre algorithme
    assert(sq >= 0); // Post-condition
    return sq;
}
```

Pré-postconditions avancées

- Les pré/postconditions peuvent être trop compliquées pour être de simples assertions ;
- Peuvent engendrer un coût supplémentaire
  - Exemple : Vérifier qu'un tableau est bien triée dans l'ordre croissant ;
  - Coût seulement acceptable durant la phase de développement
  - Utiliser les macros pour cela (ici en utilisant l'option `-DDEBUG` à la compilation) :

```
#if defined(DEBUG)
... // partie programme complexe permettant de vérifier une condition complexe;
#endif
```

- Conditions parfois difficiles à traduire dans un langage de programmation :
  - Exemple : Précondition pour le tri : la fonction de comparaison vérifie t'elle une relation d'ordre ?
  - Rajouter un commentaire dans la documentation code pour ce type de condition
- Les pré/postconditions font parti de la documentation du code.

Exercices

- Écrire une fonction qui calcule le PGCD de deux entiers a et b à l'aide de l'algorithme d'Euclide :

$$\left\{\begin{array}{ll} \text{Tant} & \text{que } b \neq 0 \\ & t \leftarrow b \\ & b \leftarrow a \mod b \\ & a \leftarrow t \\ \text{retourner} & a \end{array}\right.$$

- Écrire une fonction calculant la factorielle d'un entier :
  - de manière itérative
  - de manière récursive

Overview

Les conteneurs : Définition

Définition

Valeur contenant d'autres valeurs

Exemple : Tableaux, listes, arbres, dictionnaires, ...

Propriétés

- Réservation et libération de la mémoire gérée par le conteneur ;
- Valeurs contenues accessibles en lecture ou écriture ;
- Des itérateurs permettent de parcourir les valeurs contenues.

Conteneurs C++

- Liste d'initialisation
- Tableaux statiques
- Tableaux dynamiques
- Listes
- Ensembles
- Dictionnaires
- Queues
- Tas

Les itérateurs

Définition

Type permettant de parcourir les valeurs contenus dans un conteneur

Plusieurs types d'itérateurs avec les méthodes associées dans les conteneurs :

Type	Lecture	Écriture	Suivant	Précédent	Début	Fin
`iterator`	Oui	Oui	Oui	Non	`c.begin()`	`c.end()`
`const_iterator`	Oui	Non	Oui	Non	`c.cbegin()`	`c.cend()`
`reverse_iterator`	Oui	Oui	Non	Oui	`c.rbegin()`	`c.rend()`
`reverse_const_iterator`	Oui	Non	Non	Oui	`c.crbegin()`	`c.crend()`

- Les méthodes de type `end` "pointent" après le dernier élément ;
- L'opérateur `++` permet de passer à la valeur suivante (ou précédente) ;
- L'opérateur `!=` testent si deux itérateurs ne pointent pas sur la même valeur ;
- On accède à la valeur "pointée" par l'itérateur avec l'opérateur `*`.

```
for (auto iter = c.begin(); iter != c.end(); ++iter ) {
    *iter = 1;
}
```

Les itérateurs (suite)

Selon le type de conteneur les itérateurs peuvent être :

- **Uni-directionnels** : opérateur ++ seulement
- **Bi-directionnels** : opérateur ++ et --
- **Aléatoires** : opérateur ++,texttt–,+= et -=

```
for (auto iter=tab.begin(); iter != tab.end(); iter += 2) ...
```

Depuis C++ 11, il existe pour les conteneurs (et tout type possédant des méthodes begin et end) trois autres formes de boucle for :

- Première forme : pour lire les valeurs d'un conteneur

```
for (typeContenu itValue : conteneur ) { ... typeContenu a = itValue; ... }
```

- Seconde forme : pour lire/écrire les valeurs d'un conteneur

```
for (typeContenu &itValue : conteneur ) { ... itValue = 2; ... }
```

- Troisième forme : pour lire ou/et écrire les valeurs d'un conteneur + indices (C++ 20)

```
for (type var = value; typeContenu &itValue : conteneur ) { ... itValue = 2; ... }
```

Copie et déplacement

- À partir du C++ 11, on peut déplacer des valeurs (ou bien sûr les copier !) ;
- **La copie** duplique une valeur contenue dans une variable dans une autre variable ;
- **Le déplacement** permet à une variable v_1 de "voler" la valeur d'une autre variable v_2 : la variable v_2 a perdu sa valeur !
- `std::move` permet de spécifier un déplacement plutôt qu'une copie :

```
int v2 = v1; // Copie dans v2 de la valeur dans v1
int v3 = std::move(v1); // Déplacement dans v3 de la valeur dans v1
```

- Si le déplacement n'a aucun intérêt pour des valeurs simples comme un entier, un réel, etc., il en a beaucoup pour les conteneurs ou les structures contenant un grand nombre de données ;
- Exemple :
  - La copie d'une liste se fera en $O(n)$ (n = nombre d'éléments dans la liste)
  - Le déplacement d'une liste se fera en $O(1)$!

Copie et déplacement

- Un déplacement a automatiquement lieu au retour d'une fonction si et seulement si la valeur retournée est locale à la fonction ;
- On peut spécifier qu'un paramètre d'une fonction soit une valeur en déplacement en préfixant l'argument par un double &
- Exemple :

```
std::list<double> calculeTachesPrioritaires()
{
    std::list<double> taches;
    ...
    return taches;
}

// On perdra la tâche à l'exécution de la fonction
void executerTacheUnique( Tache&& tache )
{
    ...
}

// La liste retournée est déplacée dans la variable tachesPrioritaires
auto tachesPrioritaires = calculeTachesPrioritaires();
```

Les listes d'initialisation

Type : `std::initializer_list<type>`
Représente un ensemble de valeurs servant à :

- Initialiser d'autres conteneurs avec des valeurs ;
- Initialiser une structure (voir plus loin) ;
- Simplement définir un ensemble de valeurs temporaires

Une liste d'initialisation est définie par des valeurs de même type compris entre accolades.

```
std::initializer_list<double> l = { 1.5, 3.4, -2.6, 3.0 };
```

En particuliers, cela permet de pouvoir itérer sur un ensemble de valeurs de même type :

```
std::uint64_t prod = 1;
for (auto prime : {2,3,5,7,11,13,17,19}) {
    prod *= prime;
}
```

Les tableaux statiques en C++

- Tableau statique existe en natif ;
- Mais déconseillé d'utiliser ;
- Plutôt utiliser la composante `array` de la std ;
- Plus sûre et aussi bien optimisée
- Défini le type `std::array<type valeurs>`

Plusieurs façons d'initialiser un tableau statique :

```
std::array<type,N> array1;
std::array array2{value1, value2, value3,...}; // C++ 17
std::array<type,N> array3{value1, value2, value3,...,valueN};
```

Accéder en lecture/écriture à un élément du tableau : opérateur `[]`. Attention, indice comment à zéro !

```
array1[3] = 4; // Ecrire 4 à la quatrième position
```

Il est possible de déclarer un tableau statique contenant des tableaux statiques :

```
std::array<std::array<double,3>,3> matrice3x3 = { std::array{ 1., 0., 0. },
                                                  { 0., 1., 0. },
                                                  { 0., 0., 1. } };
matrices3x3[0][1] = -1;
```

Tableaux statiques : opérations possibles

Pour un tableau `a` :

- `a.at(indice)` : Accèder à un élément en lecture/écriture avec vérification de l'indice
- `a.front()` : Accéder en lecture/écriture au premier élément du tableau
- `a.back()` : ACcèder en lecture/écriture au dernier élément du tableau
- `a.data()` : Retourne l'adresse du premier élément du tableau
- `a.empty()` : Retourne vrai si le tableau ne contient aucun élément, faux sinon
- `a.size()` : Retourne le nombre d'éléments contenus dans le tableau
- `a.fill(value)` : Remplit le tableau avec la valeur `value`
- `a.swap(a1)` : Echange les valeurs entre le tableau `a` et `a1`

Il est également possible de comparer lexicographiquement deux tableaux statiques `a` et `a1` :

```
if (a < a1) ...
if (a == a1) ...
```

Remarque : Lors du développement, il est conseillé de rajouter l'option `-D_GLIBCXX_DEBUG` ave g++ pour déboguer notre utilisation de la librairie std.

Exercice

Calcul de l'aire signée

- Définir à l'aide d'`using` le type point et le type vecteur comme des tableaux statiques contenant deux cœfficients
- Définir à l'aide d'`using` le type triangle comme un tableau statique contenant trois points
- Définir dans le programme principal un point $p(0,0)$ et un triangle $T = \{(-1,-1),(1,-1),(0,1)\}$
- Calculer les vecteurs $u_1 = pT[0], u_2 = pT[1], u_3 = pT[2]$
- Calculer le produit croisé ($u \times v = u[0].v[1] - u[1].v[0]$) des trois vecteurs u_i avec p, deux à deux
- Vérifier que les trois produits obtenus ont le même signe
- Si oui, afficher que le point p est bien dans le triangle.

Tableaux dynamiques

- La composante `vector` de la bibliothèque standard propose un type gérant les tableaux dynamiques ;
- Type optimisé gérant dynamiquement la mémoire avec des techniques d'allocation élaborées :
- Deux notions à bien distinguer :
  - **La capacité** : La place mémoire allouée pour potentiellement contenir un certain nombre c, appelé capacité d'éléments ;
  - `u.capacity()` renvoit le nombre d'éléments potentiels alloué ;
  - **La taille** : Nombre d'éléments contenus effectivement dans le tableau.
  - `u.size()` renvoit le nombre d'éléments effectifs contenus dans le tableau ;
- La taille diffère en général de la capacité : `u.size()` $\neq$ `u.capacity()` ;
- `u.reserve(n)` permet d'allouer une place mémoire pouvant contenir potentiellement n éléments :
  - Si `n` $\geq$ `u.size()` : déplace les éléments contenus dans `u` dans la nouvelle zone mémoire réservée ;
  - Si `n` $<$ `u.size()` : déplace les n premiers éléments contenus dans la nouvelle zone mémoire et perd le reste.

Services associées aux tableaux dynamiques

- Rajout/Suppression valeur
  - `u.push_back(variable);` : Copie la valeur contenue dans `variable` à la fin du tableau ;
  - `u.emplace_back(valeur)` : Rajoute la valeur `valeur` à la fin du tableau ;
  - `u.pop_back();` : Élimine le dernier élément du tableau ;
  - `u.shrink_to_fit();` : Modifie la taille réservée par le tableau pour que la capacité soit égale à la taille actuelle du tableau
  - `u.swap(v)` : Echange le contenu de `u` avec celui de `v` ($\mathcal{O}(1)$)
- Accès valeur
  - `u[i]` accède sans contrôle d'indice (sauf option `-D_GLIBCXX_DEBUG` avec gcc) à la $i^{\text{ème}}$ valeur du tableau
  - `u.at(i)` accède avec contrôle d'indice à la $i^{\text{ème}}$ valeur du tableau
  - `u.front()` : Accède au premier élément du tableau ;
  - `u.back()` : Accède au dernier élément du tableau ;
  - `u.data()` : Retourne l'adresse mémoire du premier élément du tableau ;

Note : Le déplacement d'un `vector` est en $\mathcal{O}(1)$. Il n'est donc pas pénalisant à partir de C++11 de renvoyer un tableau dynamique local en retour d'une fonction !

Exemple d'utilisation d'un tableau dynamique

```
std::vector<std::int64_t> primes;
primes.reserve(nbPrimes);
primes.emplace_back(2);
if (nbPrimes > 1) primes.emplace_back(3);
std::int64_t k = 1;
while (primes.size() < nbPrimes) {
    std::int64_t p1 = 6*k-1;
    std::int64_t p2 = 6*k+1;
    k = k + 1;
    for (auto p : primes) {
        if ( (p1 > 0 ) and (p1%p == 0) ) p1 = 0;
        if ( (p2 > 0 ) and (p2%p == 0) ) p2 = 0;
        if ( (p1 == 0) and (p2   == 0) ) break;
    }
    if (p1 > 0) primes.push_back(p1);
    if ( (p2 > 0) and (primes.size() < nbPrimes) ) primes.push_back(p2);
}
std::cout << "Voici les " << nbPrimes << " premiers nombres premiers :  " << std::endl;
for (auto p : primes ) std::cout << p << " ";
std::cout << std::endl;
```

Exercice sur les tableaux dynamiques

- Définir un polynôme comme tableau dynamique des cœfficients de ses monômes (réels) ;
- Définir une fonction qui crée et réserve en mémoire un polynôme de degré n ;
- Écrire une fonction qui renvoie le degré d'un polynôme ;
- Écrire une fonction qui évalue un polynôme en un point x ;
- Écrire un opérateur faisant la sommation de deux polynômes ;
- Écrire un opérateur faisant la multiplication de deux polynômes ;

Les listes

- Listes doublement chaînées `std::list<Type>` dans la composante `list` de la std ;
- Listes simplement chaînées `std::forward_list<Type>` dans la composante `forward_list` de la std (C++ 11).

Méthodes communes

- `l.front();` : Accès à la première valeur ;
- `l.empty();` : Vrai si aucune valeur dans la liste ;
- `l.clear();` : Supprime toutes les valeurs de la liste ;
- `l.push_front(var);` : Insère valeur de `var` au début ;
- `l.emplace_front(params...);` : Insère valeur dépendant de params au début ;
- `l.pop_front();` : Supprime 1^{er} élément ;
- `l.swap(l2);` : Echange les deux listes (de même type) ;
- `l.merge(l2);` : Fusionne la liste l2 à la fin de la liste l ;
- `l.remove(valeur);` `l.remove_if(predicat);` : Enlève des éléments à la liste égale à la valeur ou vérifiant le prédicat ;
- `l.reverse();` : Renverse l'ordre des valeurs ;
- `l.unique();` : Enlève les valeurs dupliquées consécutives ;
- `l.sort();l.sort(comp);` : Trie la liste dans l'ordre croissant ou suivant la fonction de comparaison comp ;

Les listes doublement chaînées

Initiation d'une liste doublement chaînée :

- `std::list<Type> l;` : Construit une liste vide ;
- `std::list<Type> l = {val1, val2, ..., valN};` : Construit une liste contenant intialement les valeurs val1 à valN.

Services propres aux listes doublement chaînées :

- `l.back();` : Accède au dernier élément ;
- `l.size();` : Retourne le nombre d'éléments contenus ;
- `l.insert(iter,value)` ou `l.insert(iter, {value1, value2, ..., valueN})` : Insère à la position de l'itérateur iter la (les) valeurs passée(s) en paramètre ;
- `l.emplace(iter, params...);` : Construit une valeur dépendant des paramètres passées à la position donnée par iter ;
- `l.push_back(var);` : Rajoute en dernier élément la valeur contenue dans var ;
- `l.emplace_back(var, params... )` : Construit en dernier élément une valeur dépendant des paramètres params
- `l.pop_back();` : Supprime le dernier élément ;

On peut également comparer deux listes à l'aide des opérateurs de comparaison ($<, <=, ==, >, >=, !=$).

Les listes simplement chaînées

Initiation d'une liste doublement chaînée :

- `std::forward_list<Type> l;` : Construit une liste vide ;
- `std::forward_list<Type> l = {val1, val2, ..., valN};` : Construit une liste contenant intialement les valeurs val1 à valN.

Méthodes propres aux listes simplement chaînées :

- `l.insert_after(iter, val);` ou `l.insert_after(iter,{val1, ..., valN});` : Insère la (les) valeurs après la position de l'itérateur iter ;
- `l.emplace_after(iter,params...);` : Insère une nouvelle valeur dépendant de params après la position de l'itérateur iter ;

On peut également utiliser les opérateurs de comparaison.

Les dictionnaires

C++ propose deux types de dictionnaires :

- `std::map<TypeClef,TypeValeur>` trouvé dans la composante `map` de la std :
  - Utilise des clefs qui peuvent être comparées et trie les clefs au fur et à mesure ;
  - Insère une valeur en $\mathcal{O}(\log_2(N))$;
  - Trouve une valeur en $\mathcal{O}(\log_2(N))$;
- `std::unordered_map<TypeClef,TypeValeur>` trouvé dans la composante `unordered_map` de la std depuis le C++ 11 :
  - Utilise des clefs hashables (on peut définir sa fonction de hashage) ;
  - Insère une valeur en $\mathcal{O}(1)$
  - Trouve une valeur en $\mathcal{O}(1)$

Les deux dictionnaires proposent peu ou prou les mêmes services !

Exemple basique d'utilisation

```
std::map<std::string,std::int32_t> badges = { {"John", 12304}, {"Eric", 3204}, {"Mathew", 1320} };

badges["Paul"] = 14503;

auto iter = badges.find("Marie");
if (iter == badges.end())
  std::cout << "Marie n'existe pas" << std::endl;
else
  std::cout << "Le badge de Marie est le " << iter->second << std::endl;

iter = badges.find("Eric");
if (iter == badges.end())
  std::cout << "Marie n'existe pas" << std::endl;
else
  std::cout << "Le badge d'Eric est le " << iter->second << std::endl;

for (auto const& values : badges )
  std::cout << values.first << " a pour badge " << values.second << std::endl;
```

Overview

Généralités sur les pointeurs

- Les pointeurs représentent une simple adresse mémoire ;
- Il existe des pointeurs natifs au langage avec la syntaxe suivante : `Type *pt_var;` qui désigne un pointeur sur des valeurs de type `Type`. Il est déconseillé sauf si nécessaire d'utiliser ces pointeurs ;
- Deux composantes de la std proposent des pointeurs :
  - `std::shared_pointer<Type>` : Pointeur "partagé" sur un type ;
  - `std::unique_pointer<Type>` : Pointeur "unique" sur un type ;
- nullptr : Le pointeur "null", indiquant qu'un pointeur ne pointe sur aucune valeur ; Possède son propre type (`nullptr_t`).
- Les pointeurs non natifs gèrent la réservation et la libération mémoire automatiquement contrairement aux pointeurs natifs ;
- Accès à la valeur à l'aide du symbole `*` : `*pointeur` renverra la valeur pointée par pointeur ;
- Pas d'arithmétiques de pointeur sur les pointeurs non natifs.

Les pointeurs ”partagés”

Pointeur permettant de partager la valeur pointée avec d'autres pointeurs partagés.
Mode de fonctionnement :

- Un pointeur partagé est créé avec une nouvelle valeur, réservée et initialisée, pointée par ce pointeur ;
- On peut copier le pointeur partagé dans d'autres pointeurs partagés qui pointent sur la même valeur ;
- La valeur est détruite et la mémoire réservée libérée lorsque plus aucun pointeur partagé pointe sur cette valeur ;

Exemple d'utilisation : Gérer un graphe où on peut rajouter ou enlever des arêtes.
Exemple de code

```
auto pt_value = std::make_shared<std::complex<double>>( 1., -3.); // Pointeur sur valeur complexe 1.-3.i
auto pt_val2  = pt_value;// pt_value et pt_val2 pointent sur la même valeur.
pt_value = nullptr; // pt_value ne pointent plus sur une valeur.
pt_val2  = std::make_shared<std::complex<double>>(0.,1.); // pt_val2 pointe sur une nouvelle valeur
// La valeur complexe est détruite car plus aucun pointeur ne la pointe.
```

Les pointeurs ”uniques”

Pointeur garantissant que la valeur pointée ne peut pas être pointée par un autre pointeur (unique).
Mode de fonctionnement :

- Un pointeur unique est créé avec une nouvelle valeur, réservée et initialisée, pointée par ce pointeur ;
- On peut déplacer la valeur pointée vers un autre pointeur unique. Le pointeur initial est dans un état non défini.
- La valeur est détruite et la mémoire réservée libérée lorsque le pointeur unique pointant sur cette valeur est détruit ;

Exemple d'utilisation : Gérer un contexte, une fenêtre graphique, etc.
Exemple de code

```
auto pt_value = std::make_unique<std::complex<double>>( 1., -3.); // Pointeur sur valeur complexe 1.-3.i
auto pt_val2  = std::move(pt_value);// pt_value maintenant indéfini et pt_val2 pointe sur la même valeur.
pt_val2  = std::make_shared<std::complex<double>>(0.,1.); // pt_val2 pointe sur une nouvelle valeur
// La valeur complexe est détruite car plus aucun pointeur ne la pointe.
```

Conversion de type

- Il arrive parfois qu'on veuille convertir une valeur en un autre type, par exemple un réel en un entier.
- C++ propose trois types de conversions :
  - La conversion impérative "à la C" : Convertit la valeur sans vérification à l'aide du langage de base.
    Syntaxe : (*Type*)*variable ou valeur* ;
  - La conversion impérative "à la C++" : Convertit la valeur sans vérification à l'aide du langage de base ou si un "constructeur" le permettant existe.
    Syntaxe : *Type*(*variable ou valeur*) ;
  - La conversion statique : Le C++ vérifie que la conversion est valide à la compilation.
    Syntaxe : `static_cast`<*Type*>(*variable ou valeur*) ;
  - La conversion dynamique : Ne marche que sur des pointeurs ou des références. Valide la conversion à l'exécution.
    Syntaxe : `dynamic_cast`<*Type*>(*variable ou valeur*) ;

Remarque : Le dernier type de conversion n'est intéressant que pour la programmation objet. On la traitera au moment du chapître sur la programmation orienté objet.

Exemples de conversion valides et non valides

```
using dcomplex=std::complex<double>;
dcomplex z;
std::uint32_t uval32 = 3;

z = (dcomplex)uval32;// Valide, z vaut 3. + 0.i

z = dcomplex(uval32);// Valide, z vaut 3. + 0.i

z = static_cast<dcomplex>(uval32);// Valide, z vaut 3. + 0.i

z = 3. -2.i;
uval32 = (std::uint32_t)z;// Impossible. Erreur de compilation !

uval32 = (std::uint32_t&)z;// Possible mais non valide, uval32 vaut 0 !

uval32 = static_cast<std::uint32_t&>(z);// Possible mais non valide, erreur de compilation !
```

Overview

Définition des structures

Définition en C++

Définition d'un nouveau type contenant un ensemble de valeurs de types hétérogènes et de fonctions (méthodes) ;

Syntaxe

```
struct <NomStucture> {
  <Type> <NomChamps1>;
  ...
  <déclaration methode1>;
  ...
};
```

Note : Une classe étant en fait une structure "privée", on verra les méthodes dans la partie programmation objet du cours.
Exemple :

```
struct Matrice {
  std::int32_t numberOfRows;
  std::vector<double> coefficients;
};
```

Accès aux champs des structures et structures récursives

L'accès aux champs/attributs d'une valeur de type structure :

- se fait à l'aide du symbole . si une variable contient la valeur ;
- se fait à l'aide du symbole -> si un pointeur pointe sur la valeur.

Exemple :

```
Matrice mat;
mat.coefficients = std::move(std::vector<double>(100, 0.));
mat.numberOfRows = 10;

auto pt_mat = std::make_unique<Matrice>();// Création pointeur unique
pt_mat->coefficients = std::move(std::vector<double>(100, 0.));
pt_mat->numberOfRows = 10;
```

Possible de créer des structures récursives à condition que l'autoréférence se fasse via un pointeur :

```
struct ListNode {
  double valeur;
  std::unique_ptr<ListNode> nextNode;
};
```

Initiation des valeurs de type structure

Plusieurs façons d'initialiser une valeur de type structure :

- Attribut par attribut (voir transparent précédent pour exemple)
- Par liste d'initialisation (C++ 11) :

```
Matrice mat{ 10, std::vector<double>( 100, 0.); };
```

- Par attributs nommés (C++ 20) :

```
Matrice mat3{ .numberOfRows = 10 };// On n'initialise que certains champs
```

Il est également possible (et encouragé !) de donner des valeurs par défaut aux attributs d'une structure :

```
struct OctTreeNode {
    std::array<double,6> boundingBox{0., 0., 0., 0., 0., 0.};
    std::array<std::shared_ptr<OctTreeNode>,8> children{nullptr,nullptr,nullptr,nullptr,
                                                        nullptr,nullptr,nullptr,nullptr};
};
```

Exercice structure

Dessin "vectoriel" :

- Définir la structure `Point` contenant deux champs réels x et y ;
- Définir une fonction `buidPoint` prenant deux réels en paramètres et renvoyant un pointeur partagé sur un point ;
- Définir une structure `Arete` contenant deux champs `deb` et `fin` qui sont des pointeurs partagés sur des valeurs de type `Point`.
- Écrire un programme principal qui :
  - Construit une arête a_1 de coordonnées $(-1, 0)$ et $(0, 0)$;
  - Construit une arête a_2 de coordonnées a_1.fin et $(1, 0)$;
  - Afficher a_1 et a_2 (On construira des opérateurs de flux pour cela) ;
  - Rajouter 1 à l'ordonnée du point a_2.deb ;
  - Afficher a_1 et a_2 ;
  - Supprimer 1 de l'abcisse de a_1.fin et de a_1.deb
  - Modifier les sommets de a_1 : a_1.fin $= a_1$.deb et a_1.deb $= (0, -1)$.
  - Afficher a_1 et a_2.

Overview

La gestion des erreurs en C++

- Déjà vu l'instruction `assert` qui permet en phase de développement de détecter les erreurs de programmation ;
- Possibilité de gestion "à la main" en retournant des codes d'erreurs (gestion à la "C") ;
- Néanmoins pas possible de retourner un code d'erreur dans certains cas :
  - Lors de la construction d'une valeur (qu'on verra dans la partie programmation objet) ;
  - Lors de l'exécution d'une surcharge d'opérateur qui ne peut renvoyer que des types de valeurs spécifiques (Exemple : opérateur + pour concaténer deux chaînes de caractères) ;
- De plus, gestion pouvant être lourde à gérer :
  - Devoir remonter un code d'erreur jusqu'à la fonction pertinente ;
  - S'assurer qu'un code d'erreur a une signification unique ;
  - Pouvoir associer un message à chaque code d'erreur ;
  - Maintenant généralement difficile à cause des deux derniers points.
- Le C++ propose également de gérer les erreurs à l'aide d'exception (comme d'autres langages modernes) ;
- Attention : La gestion des exceptions ne règle pas tous les problèmes de conception !

Les exceptions

- Permet de gérer des erreurs dûes à des situations **exceptionnelles** ;
- Le principe :
  - Une fonction "émet" une erreur ;
  - Tant que cette erreur n'est pas "attrapée" on remonte la pile d'appel ;
  - Si une fonction attrape une exception, elle peut la gérer totalement ou partiellement et elle peut également la reémettre ;
  - Si aucune fonction ne rattrape l'exception, le programme s'arrête en affichant l'erreur ;
- Exemples de situation : Situation/Traitement possible
  - Le disque dur est plein/On sauvegarder sur un disque dur auxiliaire ;
  - Impossible d'accéder à un serveur/Tentative connection sur serveur mirroir ;
  - ...
- L'exception émise peut être un simple entier, une chaîne de caractère, ...ou une valeur de type `std::exception` !
- Plusieurs types d'exceptions sont déjà définies dans la composante `stdexcept` de la `std`.

Syntaxe C++

- On utilise le mot clef `throw` pour émettre une exception : `throw valeur;`
- Le bloc d'instruction où est suceptible d'être émise l'exception est précédé de `try` ;
- On utilise à la suite de `try` le mot clef `catch` suivi d'un bloc d'instruction pour rattraper et traiter une exception :`catch(Type& excp) { ...}` ;
- On peut mettre plusieurs `catch` à la suite pour attraper plusieurs types d'exceptions.

Fonction émettrice

```
auto green(double k, double r)
{
  if (r <= 1.E-10 ) throw "Distance trop petite"s;
  return std::polar(1./r, k*r);
}
```

Partie réceptrice du code appelant :

```
try {
  double r = std::sqrt(x*x + y*y + z*z);
  auto kernel = green(k, r);
}
catch(std::string& err ) {
  std::cerr << "Impossible de calculer le noyau :"
            << err << std::endl;
  throw err;
}
```

Exceptions pré-définies

Exceptions prédéfinies dans la composante stdexcept de std :

- `std::logic_error` : violations de préconditions logiques ;
- `std::invalid_argument` : argument passé à une fonction n'est pas valide ;
- `std::domain_error` : argument passé à une fonction n'est pas dans le domaine de définition ;
- `std::length_error` : tentative de réservaton supérieure à la taille maximale permise ;
- `std::out_of_range` : argument n'est pas compris dans un certain intervalle ;
- `std::runtime_error` : erreur qui ne peut survenir que lors d'une exécution
- `std::range_error` : erreur d'intervalle durant un calcul interne ;
- `std::overflow_error` : erreur dûe à un calcul produisant un dépassement arithmétique ;
- `std::underflow_error` : erreur dûe à un calcul produisant un nombre trop petit pour être représentable (underflow) ;

Note 1 : `std::domain_error` moins spécialisée que `std::invalid_argument` et pas utilisé dans la std.
Note 2 : Toutes ces exceptions peuvent être rattrapées indifféremment à l'aide d'une exception de type `std::exception` (qui peut être aussi étendue par l'utilisateur)

Exemple utilisation d'une exception prédéfinie

Fonction émettrice

```
auto green(double k, double r)
{
  if (k <= 0)
    throw std::range_error("k doit etre positif");
  if (r <= 1.E-10 )
    throw std::domain_error("Rayon trop petit");
  return std::polar(1./r, k*r);
}
```

Partie réceptrice du code appelant :

```
try {
  double r = std::sqrt(x*x + y*y + z*z);
  auto kernel = green(k, r);
}
catch(std::domain_error& err ) {
  std::cerr << "Erreur domaine de définition :"
            << err.what() << std::endl;
}
catch(std::exception& err ) {
  std::cerr << "Erreur calcul noyau :"
            << err.what() << std::endl;
}
```

Bonne utilisation des exceptions

- Mécanisme de rattrapage exception assez lourde en temps CPU ;
- L'utilisation des exceptions doit rester exceptionnelle ;
- Préfarable d'avoir un gros bloc d'instruction pour le try :

Mauvaise utilisation

```
for (int i=0; i<n; ++i)
    try {
      base[i] = normalize(u[i]);
    }
```

Bonne utilisation

```
try {
  for (int i=0; i<n; ++i)
      base[i] = normalize(u[i]);
}
```

Les entrées-sorties

- On a déjà vu comment afficher du texte à l'aide de `std::cout` ;
- Pour afficher du texte, on utilise l'opérateur de flux sortant $<<$;
- On peut également demander à un utilisateur de rentrer du texte avec `std::cin`
- Pour rentrer des données, on utilise l'opérateur de flux entrant $>>$;

Exemple :

```
std::string nom;
std::cout << "Rentrez votre nom...";
std::cin >> nom;
std::cout << "Bonjour " << nom << std::endl;
```

- En fait, principe généralisé aux fichiers formatés.
- En C++, il existe un type de base pour tout type de sortie : `std::ostream` et un type de base pour tout type d'entrée : `std::istream` ;
- Toute fonction ayant pour paramêtre un `std::ostream` pourra selon l'argument passé soit afficher dans un terminal soit écrire dans un fichier ;
- Toute fonction ayant pour paramêtre un `std::istream` pourra selon l'argument passé soit demander de rentrer les données au clavier soit lire un fichier formaté.

Ouvrir des fichiers en écriture

On utilise pour cela la composante `fstream` de la std

Ouvrir un fichier en écriture

- Syntaxe `std::ofstream fich(<nom fichier>[, <mode>]);`
- Si `mode` est omis, ouverture fichier formaté ;
- Si `mode` est mis égal à `std::ios::binary`, ouverture fichier binaire ;
- Pour écrire en binaire, utilise la méthode `write` ou opérateur de flux $<<$
- Pour écrire en formaté, utilise opérateur de flux $<<$;

Exemple : Écrire un fichier contenant un tableau de réels `u` : nombre de coefficients + coefficients

En formaté :

```
std::ofstream fich("sortie.txt");
if (fich.is_open() == false) exit(EXIT_FAILURE);
fich << u.size() << std::endl;
for (auto const& val : u ) fich << val << " ";
fich.close();
```

En binaire :

```
std::ofstream fich("sortie.dat",std::ios::binary);
if (fich.is_open() == false) exit(EXIT_FAILURE);
fich << u.size();
fich.write(u.data(), u.size()*sizeof(double));
fich.close();
```

Ouvrir des fichiers en lecture

On utilise pour cela la composante `fstream` de la std

Ouvrir un fichier en lecture

- Syntaxe `std::ifstream fich(<nom fichier>[, <mode>]);`
- Si `mode` est omis, ouverture fichier formaté ;
- Si `mode` est mis égal à `std::ios::binary`, ouverture fichier binaire ;
- Pour lecture en binaire, utilise la méthode `read` ou opérateur de flux $>>$;
- Pour lecture en formaté, utilise opérateur de flux $>>$;

Exemple : Lire un fichier contenant un tableau de réels pour initialiser un vecteur `u`

En formaté :

```
std::ifstream fich("sortie.txt");
if (fich.is_open() == false) exit(EXIT_FAILURE);
fich >> dim; v.resize(dim);
for (auto& val : v ) fich >> val;
fich.close();
```

En binaire :

```
std::ifstream fich("sortie.dat",std::ios::binary);
if (fich.is_open() == false) exit(EXIT_FAILURE);
fich >> dim; v.resize(dim);
fich.read(v.data(), dim*sizeof(double));
fich.close();
```

Formatage des nombres réels

Pour écriture dans un fichier formaté ou sur terminal, on peut formater la sortie à l'aide de la composante `iomanip` de la std :

- Ces formats se font dans le flux de sortie ;
- `std::setw(n)` réserve n caractères pour afficher la prochaine valeur ;
- `std::setfill(c)` remplit les caractères non remplit par la valeur afficher avec le caractère c ;
- `std::setprecision(n)` donne le nombre de chiffres à afficher après la virgule ;

Exemple :

```
constexpr long double pi{std::numbers::pi_v<long double>};
std::cout << std::setw(19) << std::setprecision(0) << pi << std::endl;
std::cout << std::setw(19) << std::setfill('_')  << std::setprecision(4) << pi << std::endl;
std::cout << std::setw(4) << std::setprecision(4) << pi << std::endl;
std::cout << std::setprecision(20) << pi << std::endl;
```

```
                  3
_____________3.142
3.142
3.1415926535897932385
```

Exercice

- Utilisez les opérateurs `|`, `+` et `*` pour définir le produit scalaire, la somme et l'homothétie de vecteurs (définis comme des `std::vector`) ;
- Écrire une fonction d'orthonormalisation d'une famille de vecteur u_i $(i = 0, N-1)$ (supposé libre) utilisant l'algorithme de Gram-Schmidt :
  - $v_0 = \frac{u_0}{\|u_0\|}$;
  - Pour $i \leftarrow 1$ à $N-1$:
    - $v_i = u_i - \sum_{j=0}^{i-1} (u_i|v_j) v_j$
    - $v_i \leftarrow \frac{v_i}{\|v_i\|}$
  - v_i, $(i = 0, N-1)$ possède une base orthonormale générant la famille de vecteur u_i.
- Gérer les erreurs éventuelles rencontrées à l'aide du mécanisme des exceptions (exemple : Quand la famille passée est non libre)
- Option : Sauvegarder la base dans fichier binaire ou formaté et écrire fonction pour la relire.

Overview

Fonctions : déduction du type de la valeur de retour

Deux déductions possibles en C++

- Le type de la valeur retournée déclarée : Retourne liste d'initialisation contenant valeurs nécessaire à valeur retournée ;

```
auto e1() -> std::array<double,3> { return {1.,0.,0.}; }
```

- Le type de la valeur retournée est simplement déclarée comme auto : le C++ déduit le type de retour de la fonction par le type retourné effectivement

```
auto e1() {  return std::array{1.,0.,0.};  }
```

Attention : Pour le second point, si la fonction possède plusieurs points de retour, il est impératif de retourner exactement le même type pour chaque retour !

Ne compile pas !

```
auto esqrt( double x ) {
  if (x >= 0) return std::sqrt(x);//double
  return 0; //entier !
}
```

Compile !

```
auto esqrt( double x ) {
  if (x >= 0) return std::sqrt(x);//double
  return 0.; //double
}
```

Fonctions : retour de valeurs multiples

- Il arrive souvent qu'une fonction doit retourner plusieurs valeurs ;
- Possibilité de passer en référence ou par pointeur (partagé ou unique) des valeurs de sortie ;
- Pointeur : attention au cas où pointeur passé est nul ;
- Ne sépare pas bien les paramètres d'entrées et les valeurs de sortie...

Exemple :

```
int divisionEuclidienne( int p, int q, int& reste ) {
  int résultat = p/q;
  reste = p - résultat * q;
  return résultat;
}
...
int quotient, reste;
quotient = divisionEuclidienne(7, 3, &reste);
```

Fonctions : retour de valeurs multiples...

- Possibilité en C++ de retourner tableau statique de N valeurs ;
- Les valeurs doivent être de type homogène ;
- Bonne séparation des paramètres d'entrée et des valeurs de sortie
- Mais tableau peu expressif sur le rôle des valeurs de sortie (qui quotient ? qui reste ?)
- Depuis C++ 17, possibilité de recevoir directement les valeurs retournée par structure ou tableau statique directement dans des variables.

Exemple :

```
std::array<int,2> divisionEuclidienne( int p, int q ) {
  int résultat = p/q;
  return {résultat, p - résultat * q};
}
...
auto res = divisionEuclidienne(7,3);// res[0]=2, res[1] = 1
auto[division, reste] = divisionEuclidienne(7,3);// A partir de C++ 17
```

Fonctions : retour de valeurs multiples...

Technique précédente marche également pour retourner plus de deux valeurs homogènes :

```
std::array<std::complex<double>,3> racinesCubiques(std::complex<double> const& z)
{
  const double pi_2s3 = 2*std::numbers::pi/3.;
  const double pi_4s3 = 4*std::numbers::pi/3.;
  double argument = std::arg(z)/3.;
  double module = std::cbrt(std::abs(z));//cbrt = Racine cubique
  return {
    std::polar(module, argument),
    std::polar(module, argument + pi_2s3),
    std::polar(module, argument + pi_4s3)
  };
}
...
auto [z1, z2, z3] = racinesCubiques(1.+1.i);
```

Fonctions : retour de valeurs multiples hétérogènes

- Plus possible d'utiliser un tableau statique !
- Pour deux valeurs, on peut utiliser le type générique `std::pair` proposé dans `utility`

Exemple :

```
std::pair<double,std::size_t> trouveLocaliseMax( std::vector<double> const& conteneur ) {
  int index = 0;
  for (std::size_t i=1; i<conteneur.size(); ++i)
    if (conteneur[i] > conteneur[index] ) index = i;
  return {conteneur[index],index};
}
...
auto [valeur,index] = trouveLocaliseMax({1., 3., 2., -5., 7., 5., 0.});
```

- Ne marche pas pour retourner plus de trois valeurs hétérogènes !

Fonctions : retour de valeurs multiples hétérogènes

- Depuis C++ 11, la std propose dans la composante `tuple` un type `std::tuple` similaire aux tuple de python ;
- Déclaration d'un `tuple` lourde à faire (on doit préciser chaque type) : `auto` bienvenue !
- La fonction `std::make_tuple` permet de créer facilement un tuple ;

Exemple :

```
auto convertionFraction( double valeur, int maxIter ) {   // Algorithme utilisant fraction continue
  std::int64_t dividende, diviseur;
  assert(valeur >= 0);
  std::int64_t partie_entiere = std::int64_t(valeur);
  double reste = valeur - partie_entiere;
  if (std::abs(reste) < 1.E-14) return std::make_tuple(partie_entiere, 1LL, 0.);
  if (maxIter == 0) return std::make_tuple(partie_entiere, 1LL, reste);
  auto [p,q,r] = convertionFraction(1./reste, maxIter-1);
  dividende = partie_entiere*p+q;
  diviseur  = p;
  reste = std::abs(valeur - double(dividende)/double(diviseur));
  return std::make_tuple(dividende, diviseur, reste);
}
...
auto [dividende, diviseur, reste] = convertionFraction(std::sqrt(2),20);
```

Fonctions : retour de valeurs multiples hétérogènes

- Solutions précédentes présentent un défaut : rôle des valeurs retournées non exprimées ;
- Une autre solution existe : utiliser une structure !
- Réception des valeurs multiples façon C++ 17 marche encore !

Exemple :

```
struct convertionFractionReturnType { std::int64_t dividende, diviseur; double reste };
convertionFractionReturnType convertionFraction( double valeur, int maxIter ) {
    std::int64_t dividende, diviseur;
    assert(valeur >= 0);
    std::int64_t partie_entiere = std::int64_t(valeur);
    double reste = valeur - partie_entiere;
    if (std::abs(reste) < 1.E-14) return {partie_entiere, 1LL, 0.};
    if (maxIter == 0) return {partie_entiere, 1LL, reste};
    auto [p,q,r] = convertionFraction(1./reste, maxIter-1);
    dividende = partie_entiere*p+q; diviseur  = p;
    reste = std::abs(valeur - double(dividende)/double(diviseur));
    return {dividende, diviseur, reste};
  }
  ...
  auto res = convertionFraction(std::sqrt(2),20);// res.dividende, res.diviseur, res.reste
```

Réception de valeurs multiples en C++ 17 (complément)

- Parfois, toutes les valeurs retournées ne nous intéresse pas.
- Par exemple, dans l'exemple de convertion en fraction, le reste peut ne pas nous intéresser
- Deux façon de faire :
  - On récupère les trois valeurs mais on rajoute une directive pour le compilateur pour ignorer le reste qui ne sert pas (C++ 17) :

```
[[maybe_unused]] auto [p,q,ignorée] =   convertionFraction(std::sqrt(2), 20);
```

  - On récupère les trois valeurs avec la fonction `std::tie` proposée depuis le C++ 11, mais on remplace par `std::ignore` la valeur qu'on veut ignorer (C++ 11 ou supérieure) :

```
std::int64_t diviseur, dividende;
std::tie(dividende, diviseur, std::ignore) = convertionFraction(std::sqrt(2),20);
```

Retour d'une variable globale vs retour d'une variable locale

Une valeur retournée par une fonction :

- est déplacée si cette valeur est une valeur locale à cette fonction ;
- est copiée si cette valeur est une valeur définie en dehors de cette fonction.

Exemple :

```
std::vector<double> operator + ( std::vector<double> const& u, std::vector<double> const& v ) {
  assert(u.size() == v.size());
  std::vector<double> w(u.size());
  for (std::size_t i=0; i<u.size(); ++i ) w[i] = u[i] + v[i];
  return w; // w est retournée par déplacement. On a un coût en O(1) pour le retour.
}
std::vector<double> operator += ( std::vector<double>& u, std::vector<double> const& v ) {
  assert(u.size() == v.size());
  for (std::size_t i=0; i<u.size(); ++i ) u[i] += v[i];
  return u; // u est retournée par copie ! On a un coût en O(n) pour le retour !
}
```

Retour par référence

- Il est possible de retourner une valeur par référence pour une fonction
- Cette valeur doit être globale (sinon bogue !) ;
- Pour le cas de l'addition sur place (+=) vue dans l'exemple précédent, on peut retourner une référence pour éviter une copie :
- Dans ce cas, la copie ne se fait pas si on reçoit la référence dans une variable référence.

```
std::vector<double>& operator += ( std::vector<double>& u, std::vector<double> const& v )
{
  assert(u.size() == v.size());
  for (std::size_t i=0; i<u.size(); ++i ) u[i] += v[i];
  return u; // u est retournée par référence !
}
...
u1 += u2; // Ok, on retourne une référence qui ne sert à rien (détruite juste après l'appel !)
auto& res = (u1 += u2);// OK, pas de copie, res fait référence à la valeur possédée par u1
auto res = (u1 += u2);// Une copie de u1 est faite dans res.
```

Paramètres nommés (C++ 20)

- En C++ 20, possible d'initialiser une structure en nommant certains paramètres de la structure ;
- Utile pour écrire une fonction avec nommage des paramètres pertinents à l'appel.

Exemple :

```
struct AxpyParameters {
  double alpha = 1;   std::size_t dim{0};
  std::vector<double> const& x; int incx{1}, offsetX{0};
  std::vector<double>      & y; int incy{1}, offsetY{0};
};

void axpy( AxpyParameters const& params ) {
  std::size_t dim = params.dim;
  if (dim==0) dim = params.x.size() - params.offsetX;
  for (std::size_t i=0; i < dim; ++i )
    params.y[i*params.incy + params.offsetY] += params.alpha * params.x[i*params.incx + params.offsetX];
}
...
axpy( {.alpha=-2., .x=u, .y=v } );// Attention à respecter ordre déclaration dans structure
```

Fonctions génériques (C++ 2020)

- Écrire plusieurs fois la même fonction pour différents types : pénible et source de bogues ;
- Depuis C++ 20, on peut déclarer un paramètre auto plutôt que de lui donner un type ;
- C'et à l'appel que le type sera défini pour ce paramètre ;
- On peut contrôler la catégorie de types permis pour un paramètre (mais on ne le verra pas dans ce cours) ;

```
// Fonction générique pour des vecteurs contenant des types multipliables et additionnables
void axpy( auto alpha, auto const& x, auto &y )
{// Op. y <- y + alpha.x où x et y sont des vecteurs, alpha un scalaire
    for (std::size_t i=0; i< x.size(); ++i) y[i] += a*x[i];
}
...
std::array fx = { 1.f, 2.f, 3.f, 4.f };
std::array fy = { 0.f,-1.f,-2.f,-3.f };
axpy(2.f, fx, fy ); // Appel axpy avec des tableaux statiques
std::vector dx = {1., 2., 3., 4.};
std::vector dy = {0.,-1.,-2.,-3.};
axpy(2.f, dx, dy); // Appel axpy avec alpha simple précision, x et y tableaux dynamiques double précision
axpy(2.f, 1.f, -2.f);// Erreur compilation !
```

Les fonctions constexpr

- On a vu qu'on pouvait définir des valeurs évaluées à la compilation ;
- C++ 11 perment de créer des fonctions appelables à la compilation pour évaluer ces valeurs ;
- Les paramêtres passés à ces fonctions doivent être constexpr pour être appelable à la compilation ;
- Ces fonctions peuvent être également appelées normalement durant l'exécution du code.
- Si elles ne sont pas appelées en dehors des constexpr, elles ne seront pas compilées pour le code exécutable
- Les fonctions mathématiques de la composante `cmath` ne sont pas constexpr (prévu pour C++ 26) ;
- Peu de compîlateur supporte les fonctions mathématiques en constexpr actuellement (seulement g++ à ma connaissance).

```
constexpr double cteExp(int n) {
  if (n==0) return 1; else {
    if (n%2==0) {
      double v = cteExp(n/2);
      return v*v;
    } else return std::numbers::e * cteExp(n-1);
  }
}
...
constexpr double cx = cteExp(7);
```

Les fonctions constexpr

- Parfois, pour une même fonction, on aimerait une implémentation différence selon qu'on calcule une expression constante ou non ;
- Depuis C++ 23, possibilité de tester si une fonction constexpr est évaluée à la compilation ou à l'exécution
- Syntaxe :
  `if consteval { ... partie exécutée à la compilation } else { ... partie exécutée à l'exécution }`

Exemple :

```
constexpr double cteExp(int n) {
  if consteval {
    if (n==0) return 1; else {
      if (n%2==0) { double v = cteExp(n/2); return v*v; }
      else return std::numbers::e * cteExp(n-1);
                              }
  } else { return std::exp(n); }
}
...
constexpr double cx = cteExp(7);// Evaluation à la compilation, appel fonction récursive
double y = 7;
double cy = cteExp(y);// Evaluation à l'exécution, appel fonction math
```

Les fonctions consteval (C++ 20)

- On aimerait forcer une fonction à être évaluée uniquement durant la compilation ;
- Depuis C++ 20, on peut pour cela déclarer une fonction `consteval` ;

```
consteval double evalExp(int n) {
  if (n==0) return 1;
  else {
    if (n%2==0) {
      double v = cteExp(n/2);
      return v*v;
    } else {
      return std::numbers::e * cteExp(n-1);
    }
  }
}
...
double ix = 7;
constexpr double cx2 = evalExp(7);
double x2 = evalExp(7);// Valeur évaluée à la compilation mais x2 non constexpr...
double x23 = evalExp(ix); // Génère une erreur de compilation !
```

Les fonctions lambda

- Il arrive qu'on ait besoin d'une fonction faisant quelques lignes pour effectuer une tâche particulière ;
- Exemple : Pour trier un tableau de vecteur selon leur norme L_2
- Les fonctions lambdas sont donc un moyen commode de définir une fonction anonyme qu'on pourra appeler ou passer en argument localement ;
- Syntaxe : [clause de capture] (liste des paramètres) mutable throw() ->type retour { corps de la fonction}
  optionnel optionnel optionnel
- La capture permet de donner une "liste" des variables externes à la fonction visibles dans le corps de la lambda fonction ;
- `mutable` et `throw()` ne seront pas abordés dans cette formation ;
- Si le type de retour n'est pas donné, le retour sera déduit du type des valeurs retournées par la fonction ;
- Une lambda fonction possède son propre type. On doit utiliser `auto` pour "stocker" la lambda fonction dans une variable ;

Exemple :

```
std::vector<std::array<double,10>> arrays;
...
std::sort(arrays, [](  std::array<double,10> const& u, std::array<double,10> const& v) {
    double sqNrmU = 0, sqNrmV = 0;
    for (int i=0; i<10; ++i) {sqNrmU += u[i]*u[i]; sqNrmV += v[i]*v[i]; }
    return sqNrmU < sqNrmV; });
```

Les captures dans les fonctions lambda

- Une valeur capturée peut être capturée par valeur (on en fait une copie) ou bien par référence ;
- Une variable précédée par un & indique que sa valeur est passée par référence ;
- Si la clause de capture est vide (`[]`), cela indique que le corps de la lambda ne peut accèder à des variables non définis localement ;
- Il est possible d'indiquer un mode de capture par défaut de toutes les variables externes utilisées par la lambda ;
- `[&]` indique que toutes les variables externes utilisées sont capturées par référence ;
- `[=]` indique que toutes les variables externes utilisées sont capturées par valeur ;
- Il est possible d'indique un mode de capture par défaut et une liste de variables capturées autrement.
  Exemple : `[=,&total]`

Exemple :

```
double maxNorm = 0;
std::vector<std::array<double,10>> arrays;
...
std::sort(arrays.begin(), arrays.end(), [&maxNorm]( std::array<double,10> const& u,
                                                    std::array<double,10> const& v) {
    double sqNrmU = 0, sqNrmV = 0; for (int i=0; i<10; ++i) {sqNrmU += u[i]*u[i]; sqNrmV += v[i]*v[i]; }
    maxNorm = std::max({maxNorm, sqNrmU, sqNrmV});  return sqNrmU < sqNrmV; });
```

Les fonctions lambda : paramètres auto (C++ 14) et affectation

- Comme pour les fonctions génériques en C++ 20, il est possible de déclarer un paramètre de type auto ;

Exemple paramètre générique et affectation à une variable

```
double evalPol( double x ) {  return 3*x*x*x - 4.*x*x + x - 5;  }
...
auto compose = [] (auto const& f, auto const& g) {
        return [f, g]( double x ) { return f(g(x)); };
                                                  };
auto f = compose(evalPol, cos);
double x = std::numbers::pi;
std::cout << "f(pi) = " << f(x) << std::endl;
```

Exercice

- Écrire la fonction $f(x) = \frac{x^3 - 3.x^2 + 2}{|x|^2 + 1}$ où x peut être tout type de valeur scalaire ;
- Dans la fonction `main`, déclarer un complexe c et une variable m à laquelle on affectera la lambda fonction $m(z) = z^2 + c$ où z est un complexe ;
- Déclarer un tableau dynamique R de N (=10) réels double précision dont on remplira les cœfficients avec $R[i] = \cos(\frac{i.\pi}{N})$
- Déclarer un tableau dynamique Z de N complexes dont les valeurs sont définies par récurrence : $Z[0] = c$ et $Z[i+1] = m(Z[i])$;
- Trier le tableau R tel que $f(R[i]) \leq f(R[i+1])$;
- Trier le tableau Z tel que $|f(Z[i])| \leq |f(Z[i+1])|$.