RÉPUBLIQUE
FRANÇAISE

Liberté
Égalité
Fraternité

ONERA

THE FRENCH AEROSPACE LAB

www.onera.fr

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

C++

Présentation et initiation au C++ 98 à 2020

Généalogie du C++

- 1969 : Première version d'Unix en assembleur
- 1969 : Langage B (interprété)
- 1971 : Langage C (pour Unix en C)
- 1980 : Langage C++
- 1983 : Standardisation ANSI du C
- 1998 : Standardisation ISO du C++ (a.k.a C++ 98)
- 2011 : Mise à jour ISO du C++ (C++ 11)
- 2014 : Mise à jour ISO du C++ (C++ 14)
- 2017 : Mise à jour ISO du C++ (C++ 17)
- 2020 : Mise à jour ISO du C++ (C++ 20)
- 2023 : Mise à jour ISO de prévu…

Caractéristiques du C++

- Langage compilé
- Multiparadigme : Structuré, orienté objet, fonctionnel
- Bibliothèque standard ISO très riche :
  - Pointeurs intelligents, chronomètres, fonctions de hashage, …
  - Tableaux statiques, dynamiques, listes, dictionnaires, queues
  - Fonctions de tris complets ou partiels, recherches rapide,…
  - Gestion chaînes de caractères ASCII, UTF8, …
  - Entrées-sorties, gestion fichier/répertoire…
  - Complexes, polynômes de Legendre, Hermite, fonction Zêta…
  - Expressions régulières
  - Gestion threads posix et versions parallèles de fonctions
  - Vues, évaluations paresseuses, etc.
- Impossible maîtriser 100% : 10% pour un débutant…

Compilateurs (gratuits !)

- Linux : g++ ou clang++ (Iso 17/20)
- Windows :
  - Msys 2 + g++/clang++ (ISO 20)
  - WSL (**W**indows **S**ubsystem For **L**inux) : voir Linux
  - Codeblock (ISO 17)
  - Visual C++ community (ISO 17, options ≠)
- Mac : homebrew + g++ ou clang++ (Iso 17)
- Androïd : C4droid (g++ ISO 20)
- Internet : `https://www.onlinegdb.com/online_c++_compiler`

Editeurs

- Atom
- Sublime text (Vérification à la volée avec Clang++)
- Visual Code
- Codeblock
- Emacs, Vim, …
- Tout éditeur de texte qui vous convient
- Evitez gedit qui rajoute des caractères de contrôle invisibles (erreur de compilation dur à voir !)

Invocation compilateur (g++/clang++)

- Mêmes options pour les deux compilateurs
- Remplacez ci—dessous g++ par clang++ si vous utilisez clang++
- Remplacer –std=c++20 par –std=c++17 si votre compilateur ne supporte pas C++ 20
- Pour développer/déboguer :
  - g++ -std=c++20 –g –pedantic –Wall –D_GLIBCXX_DEBUG -o <nom exécutable> <fichiers sources>
- Pour production/optimisation:
  - g++ -std=c++20 –march=native –O3 –DNDEBUG –o <nom exécutable> <fichiers sources>
- Un Makefile permet de s'affranchir de toutes ses options à chaque compilation !

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

Bonnes pratiques de programmation

Initiation à la qualité logicielle

Motivations

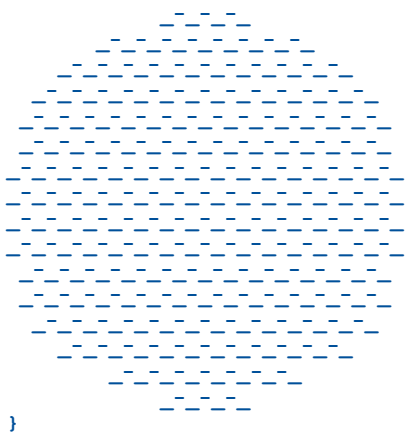
- **Vie d'un logiciel** : plus de temps à le lire qu'à programmer
- Code clair et agréable à lire : très important
- Analogie entre l'écriture d'un texte et d'un code :
  - Bien écrit
  - Bien présenté
  - Pas de fautes d'orthographe
  - Phrases bien structurées
  - Idées bien organisées et successions logiques
- Beaucoup d'énergie dépensée pour rien pour déboguer un code mal écrit et mal présenté

Exemple de « mauvais » code

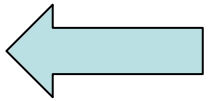
```
int k(int i)
{int rsflkj=1; if (i==1)
return rsflkj; else rsflkj = i;
return rsflkj*k(i-1);}
```

Que fait cette fonction ?

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
```



```
}
```



Concours annuel
International Obfuscated C Code
Gagnant concours 1988

https://www.ioccc.org/

Contrat-interface versus mise en œuvre d'algorithme

- **Contrat** : caractérise l'interface
  - Qu'est ce que l'algorithme est capable de produire
  - Domaine de définition de l'algorithme
  - Valeurs possibles en sortie
- **Exemple** : *racine carrée d'un réel*
  - En entrée : un réel qui doit être positif ou nul
  - En sortie : un réel qui doit être positif ou nul
- **Précondition** : Quelles conditions doivent vérifier les valeurs connues en entrée de l'algorithme ?
- **Postcondition** : Quelles conditions doivent vérifier les valeurs connues en sortie de l'algorithme ?

Assertions

- En C/C++, utilisation des assertions pour les Post/Préconditions
- Utilisation de la bibliothèque `<cassert>` en C++ (`<assert.h>` en C)
- Les assertions ne sont pas vérifiées si l'option `-DNDEBUG` a été spécifiée à la compilation
- Exemple programme C pour la racine carrée :

```
#include <cassert>
double sqrt ( double x )
{
   assert ( x>=0) ; // Précondition
   sq = . . . // Calcul de la racine qu'on stocke dans
sq
   assert ( sq >= 0) ; // Post-condition

   return sq ;
}
```

Pré/Postconditions en C++ (suite)

- Peuvent être plus que de simples assertions
- Peuvent engendrer un coût supplémentaire
  - Exemple : vérifier qu'un tableau a bien été trié dans l'ordre croissant
  - Mais seulement lors de la phase de développement
- Peuvent être difficile à traduire en C++
  - Exemple : Précondition pour le tri : l'opérateur de comparaison vérifie t'il bien une relation d'ordre ?
  - Dans ce cas, c'est au programmeur de vérifier à la main si c'est bien le cas !
  - Le rajouter en commentaire pour la documentation du code
- Les pré/postconditions font aussi parti de la documentation du code.

Caractéristiques d'un code « bien écrit »

- Être facile à lire
- Avoir une organisation logique et évidente
- Être explicite
- Soigné et robuste au temps qui passe

Être facile à lire

- Bien structuré et bien présenté
- Noms des variables et des fonctions choisis avec soin
- Bien respecter les règles d'indentation
  - Blocs d'instructions au même niveau ➔ précédés du même nombre d'espace
  - Exemple code mal indenté versus code bien indenté

```
void m( int n, float * A, float * B,
float * C) {
int i , j , k ;
for ( i = 0 ; i < n ; ++i ) {
float a = 0 . ;
for ( j = 0 ; j < n ; ++j ) {
for ( k = 0 ; k < n ; ++k ) {
a += A[ i+k*n ] *B[ k+j *n ] ;
}
C[ i+j *n ] += a ;
}
```

```
void m( int n, float * A, float * B,
float * C)
{
   int i , j , k ;
   for ( i = 0 ; i < n ; ++i )
   {
      float a = 0 . ;
      for ( j = 0 ; j < n ; ++j )
      {
         for ( k = 0 ; k < n ; ++k )
            a += A[ i+k*n ] *B[ k+j
*n ] ;
      }
     C[ i+j *n ] += a ;
   }
}
```

Organisation logique et évidente

- Notion parfois plus subjective : chacun solution ≠
- Essayer de trouver les solutions les plus simples
  - Exemple : pour afficher les nombres de 1 à 10 :
    - Faire une boucle allant de 1 à 10 pour afficher les nombres
    - Ne pas faire une boucle i allant de 9 à 0 et afficher 10-i
- Eviter d'avoir des paramètres redondants ou se déduisant d'autres paramètres

```
void
orthonogalise ( double u[3] , double nrmu ,
double v[3] )
{
   double dotuv = u[0]*v[0] + u[1]*v[1]+ u
[2]*v[2] ;
   v [0] = v[0] - dotuv*u[0]/(nrmu*nrmu) ;
   v [1] = v[1] - dotuv*u[1]/(nrmu*nrmu) ;
   v [2] = v[2] - dotuv*u[2]/(nrmu*nrmu) ;
}
```

```
void orthonogalise ( double u[3] , double v[3]
)
{
  // Calcul ||u||²
  double sqr_nrm_u = u [0]*u[0] +u[1]*u[1] +
u[2]*u[2] ;
  // Précondition vérifiant que le vecteur u
n'est pas nul.
  assert (sqr_nrm_u > 1 .E-14) ;
  double dotuv = u[0]*v[0] + u[1]*v[1] +
u[2]*v[2] ;
   v[0] = v[0] - dotuv*u[0]/sqr_nrm_u ;
   v[1] = v[1] - dotuv*u[1]/sqr_nrm_u ;
   v[2] = v[2] - dotuv*u[2]/sqr_nrm_u ;
   // Postcondition vérifiant que v orthogonal
à u
```

Le code doit être explicite

- Lorsqu'on développe des algorithmes :
  - prendre des raccourcis autorisés
  - Mais bien prendre soin de l'expliquer avec des commentaires
  - Permet de se souvenir de l'astuce plus tard et pour les autres
- Exemple
  - Afficher une matrice MxM
  - Normalement à l'aide de deux boucles
  - Or on sait que nos matrices sont triangulaires
  - Optimiser le code pour des matrices triangulaires
  - Bonne idée mais commenter pour expliquer pourquoi on procède de la sorte !

Code soigné et robuste au temps qui passe

- Ne pas s'arrêter dès qu'un code marche !
- Entretient du code important !
  - Supprimer les éléments obsolètes
  - Vérifier que les commentaires sont à jour et cohérents
- « Maintenance » du code crucial
  - Surtout lorsqu'on rencontre des bogues
- Exemple
  - Une fonction `tri` qui trie des éléments d'un tableau;
  - On remplace tri par un `tri_rapide` plus adapté qui semble fonctionné mais vous laissez la fonction `tri` dans le code;
  - Plusieurs mois plus tard, un bogue est détecté qui semble provenir du tri;
  - Analyse de la fonction `tri` pendant longtemps jusqu'à ce que vous réalisez que c'est maintenant `tri_rapide` utilisé.

Exemple de commentaires non mise à jour

```
void une_fonction ( bool continuer )
{
   // La boucle s ’arrête si i est négatif ou si continuer prend la
valeur false
   int i = 0 , j = 4 ;
   while ( continuer )
   {
      std : :cout << ”Mon code marche” << std : :endl ;
      // i += 1;
      j += 1;
      if ( j >10) continuer = true;
   }
}
```

À votre avis, pourquoi les commentaires obscurcissent le code plutôt que de l’éclairer ?

Coder proprement, ça prend du temps ?

- Ne pas confondre vitesse et précipitation !
- En fait on gagne du temps :
  - Pas si lourd à faire si on le fait dès le départ (50% du travail fait)
  - Code bien écrit : plus facile et donc plus rapide à relire
  - On passe plus de temps à relire qu'à écrire
  - Code logique et bien structuré : plus facile de retrouver des bogues
  - Plus facile à l'étendre et donc de l'améliorer.

De l'importance des commentaires

- Essentiels pour éclairer le code
- Un bon commentaire
  - Facilite la lecture du code
  - Apporte une indication sur un choix de conception
  - Explique une motivation qui ne serait pas évidente
  - Donne un exemple pour mieux comprendre ce que fait le code
- Un mauvais commentaire
  - Décrit un morceau de code qui n'existe plus
  - Explique une évidence
  - Fait plusieurs lignes pour expliquer une chose simple
  - Est un historique sur la modification des fichiers : c'est une mauvaise idée, il vaut mieux confier cela à un gestionnaire de tâche (exemple : git)

Exemple critiquable de commentaires

```
i = 0 ; // On initialise la variable i à zéro
i = i + 1 ; // On incrémente de un la variable i
// On additionne a et b et on stocke le résultat dans c
c = a + b ;
// Ci--dessous , on fait une double boucle pour afficher la matrice :
for ( i = 0 ; i < 10 ; ++i )
    std::cout << " valeur : " << i << " " ;
// Fin du for
std::cout << std::endl; // Retour à la ligne
/*
Et maintenant , on va s ' occuper de retourner la valeur de i . On
utilise pour cela
l ' instruction return à laquelle on passe la valeur de i
*/
return i ;
```

Comment nommer les choses ?

```
gfdjkgldfj = 4 ;
ezgiofdgfdljkrljl = 1 ;
gfdjkgldfj = ezgiofdgfdljkrljl +
gfdjkgldfj ;
```

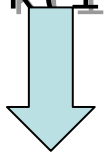
```
x = 4 ;
x += 1 ;
```

- Choisir des noms de variables pronoçables et faciles à retenir
- Choisir des noms de variables explicites pour vous et les autres
  - Par exemple, `a` moins explicite que `adresse_client`
  - Par exemple, `lf` moins explicite que `largeur_fenetre`
  - Combien d'occurrence de `a` dans le code ? Combien de `adresse_client` ?
- Eviter un nom de variable qui introduit un contre-sens
  - `matrice = 8`
  - On peut penser que c'est une matrice, mais c'est clairement un entier !
  - Imaginez que vous voyez plus loin la ligne suivante : `matrice = 4 * matrice;`
  - Que penser de cette ligne ?
- Eviter des noms de variables qui n'ont pas de sens (exemple : `plop`)
- Eviter de tricher en choisissant des noms proches d'un mot clef.
  - Exemple : `ccase, vvolatile`
- Eviter de mélanger du français et de l'anglais (exemple : `lengthChemin`)

Et pour conclure

- Voici comment arranger le premier code :

```
int k(int i)
{int rsflkj=1; if (i==1)
return rsflkj; else rsflkj =
i;
return rsflkj*k(i-1);}
```



```
long fact ( long n)
{
   assert (n>=0) ; // Précondition
   i f (n == 0) // Cas particuliers : 0 ! =
1
       return 1 ;
   long resultat = n ∗ fact(n−1) ;// n ! =
n ∗ (n−1) !
   assert ( resultat > 0) ; //
Postcondition
   return resultat;
}
```

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

Initiation au langage C++

Pour commencer…

- Bjarn Stroustrup : 70% langage,
- Expert : 60%,
- Débutant : 10%
- Quelques pages internet de référence :
  - `https://en.cppreference.com/w/`
  - `http://www.cplusplus.com/`

Un petit programme éponyme en C++

```
#include <iostream>

int main ( )
{
      std::cout << ” Hello World ! ” <<
std::endl;
      return EXIT_SUCCESS ;
}
```

- Pour afficher sur console : utilisation de `<iostream>`
- Utilisation de `std::cout` (Console Output) et des flux de sortie (<<)
- `std::endl` pour le retour à la ligne (end line)
- `EXIT_SUCCESS` : En C également pour signaler que le programme s’est passé sans accros

Convention sur les noms de variables (et autres)

- Peut contenir des caractères ASCII
- Ne doit pas contenir des espaces ou des tabulations
- Ni de ponctuations, de quotes, de symboles d'opérations, de parenthèses, brackets et accolades ni des symboles @ et ©
- Ne peut pas commencer par un chiffre
- Depuis C++ 11, peut contenir une grande partie des caractères unicodes
  - Valides : a, _a, clé, périmètre, ,
  - Invalides : 1a, }c, la clef, <c

Le type booléen

- Mot clef natif au C++ : `bool`
- Ne peut prendre que deux valeurs : `true` ou `false`
- Opérations logiques et de tests valables sur eux
- À l'affichage, affiche 0 (false) ou 1 (true) sauf si on utilise `std::boolalpha` de la bibliothèque `iomanip`

```
#include <iostream>
#include <iomanip>
int main ( )
{
	bool f 1 = (3>5); // f1 est faux
	bool f 2 = f1 || (5-7+2 == 0); // f2 est vrai
	std::cout << std::boolalpha << ”f1 : ” << f 1 << std::endl;
	std::cout << ” et f2 : ” << f2 << std::endl ;
	std::cout << std::noboolalpha << ” f1 && f2 : ” << f1 && f2 <<
std::endl ;
	std::cout << ”f1 || f2 : ” << f1 || f2 << std::endl ;
	return EXIT_SUCCESS ;
}
```

Les entiers

- Même types de base qu'en C
- Attention : char signé ou non signé selon les compilateurs/système d'exploitation;
- Type long 32 bits sous Windows, 64 bits sous Linux
- Attention au débordement d'entier !
- Eviter si possible les entiers non signés sources de bogues difficiles à trouver :

```
unsigned i , j ;
  for ( i = 1 ; i < 99 ; ++i )
  {
      for ( j = i +1 ; j >= i –1 ; ––j )
      { ...
```

- Attention également à la division entière ! 5/2 = 2 !
- Affichage grâce aux flux :

```
long s = 32769 ;
signed t = 130 ;
std::cout << ”s = ” << s << ” et t = ” << t << std::endl;
```

Utilisation de <cstdint>

- Permet de définir des entiers avec un nombre de bits précis indépendant du compilateur et du système d'exploitation

```
#include <cstdint>
int main( )
{
    std::uint8_t    byte ;   // entier non signé représenté sur 8 bits ( un
octet )
    std::int8_t     sbyte ;  // entier signé représenté sur 8 bits ( un octet )
    std::uin16_t    ush ;    // entier non signé représenté sur 16 bits ( deux
octets )
    std::int16_t    sh;      // entier signé représenté sur 16 bits ( deux
octets )
    std : :uint32_t uent ;  // entier non signé représenté sur 32 bits
( quatre octets )
    std::int32_t     ent ;  // entier signé représenté sur 32 bits ( quatre
octets )
    std::uint64_t    ulg ;  // entier non signé représenté sur 64 bits ( huit
octets )
    std::int64_t     lg ;   // entier signé représenté sur 64 bits ( huit
octets )
```

De la non utilisation des entiers non signés

```
// Recherche racine carrée d'un entier de la
// forme n² par dichotomie
std::uint32_t n² = 3249 ; // n² = 57 ²
std::uint32_t sup = n² ;
std::uint32_t inf   = 0 ;
std::uint32_t milieu = (inf+sup)/2 ;
while (milieu*milieu−n²!= 0)
{
   if (milieu*milieu−n² < 0) // Bogue ici !
   {
        inf = milieu ;
        milieu = (inf+sup)/2 ;
   }
   else
   {
        sup = milieu;
        milieu = (inf+sup)/2;
   }
}
assert (milieu*milieu == n²) ; // Postcondition
std::cout << "√" << n² << " = " <<
milieu
                << std::endl;
```

```
// Recherche racine carrée d'un entier de
// la forme n² par dichotomie
std::int32_t n²= 3249 ; // n² = 57²
assert (n²>=0);
std::int32_t sup = n²;
std::int32_t inf = 0;
std::int32_t milieu = (inf+sup)/2;
while (milieu∗milieu−n²!= 0)
{
   if (milieu*milieu−n²<0)
   {
        inf = milieu;
        milieu = (inf+sup)/2;
   }
   else
   {
        sup = milieu;
        milieu =(inf+sup)/2 ;
   }
}
assert (milieu*milieu==n²);
std::cout << " √" << n² << " = " <<
milieu
                << std : :endl ;
```

Formatage des entiers en sortie

- Utilisation de `iomanip`
- `std::setw` réserve un nombre d'espace pour afficher
- `std::fill` 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 ')
```

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

Les réels

- Comme en C, 3 types : `float, double, long double`
- Trois valeurs spéciales en plus depuis C++ 11 dans `<limits>`
  - `quiet_NaN` : Not a Number, pas d'erreur à sa première apparition
  - `signaling_NaN` : Not a Number, lève une erreur à sa première apparition
  - `infinity` : Représente l'infini

```
float pas_un_nombre = std::numeric_limits<float>::quiet_NaN();
double infini = std::numeric_limits<double>::infinity();
```

Les réels (quiet_NaN)

Toujours différents d'un autre réel, dont lui-même !

```
bool is_equal = std::numeric_limits<double>::quiet_NaN() ==
                        std::numeric_limits<double>::quiet_NaN();
std::cout << std::boolalpha << is_equal << std::endl;
```

```
false
```

std::is_nan pour tester si ce n'est pas un nombre

```
double x = 0./0.;
std::cout << std::boolalpha << "x est un nan ? " << std::isnan (x)
<< std::endl ;
```

```
x est un nan ? true
```

Les réels (infinity)

Toujours supérieur à n'importe quel nombre réel

```
#include <limits>
int main ( )
{
    float fx = std::numeric_limits<float>::max(); // valeur maximale
d'un float
    float finf = std::numeric_limits<float>::infinity();
    std::cout << std::boolalpha << fx << " < ∞ ? : " << ( fx < finf )
<< std::endl ;
    return EXIT_SUCCESS ;
```

```
3.40282e+38 < ∞ ? :  true
```

Fonctions mathématiques

Fonctions mathématiques usuelles du C dans `<cmath>`

```
float π_f = std::acos(-1.f) ;
float fx = std::cos(π_f/4.f) ;
double π = std::acos(1.);
double x = std::cos(π/4.) ;
```

Depuis C++ 11, d'autres fonctions proposées

```
double x = 3, y = 2, z = 5;
double h = std::hypot(x,y,z);// Calcul √(x²+y²+z²)
double p = std::hermite(4, x);//Calcul 16x^4 – 48x^2 + 12
double ζ = std::riemann_zeta(-1);// Calcul la fonction zeta de Riemann en -1
double sp = std::sph_bessel(2,x);// Calcul fonction de Bessel sphérique d'ordre 2
```

Les constantes prédéfinies (C++ 20)

En C++ 20, bibliothèque <numbers> proposent constantes usuelles

```
#include <cmath>
#include <numbers>
#include <iostream>
int main ( )
{
    float π_f  = std::numbers::pi_v<float>;
    double π = std::numbers::pi;
    long double π _lf = std::numbers::pi_v<long double>;
    double π¯¹ = std::numbers::inv_pi;
    std::cout << ”π_f = ” << std::setprecision(std::numeric_limits<float>::digits10+1)
<< π_f << std::endl ;
    std::cout << ”π = ” <<
std::setprecision(std::numeric_limits<double>::digits10+1) << π << std::endl;
    std::cout << ”π _lf = ” << std::setprecision(std::numeric_limits<long
double>::digits10+1) << π _lf
                    << std :endl ;
    std : :cout << ”π¯¹ = ” <<
std::setprecision( std::numeric_limits<double>::digits10+1) << π¯¹ << std : :endl ;
    return EXIT_SUCCESS ;
}
```

Les complexes

- Pas natif. On doit utiliser la bibliothèque `<complex>`
- Générique : complexe avec entiers, float, double, etc.
- Fonctions usuelles compris
- Attention : std::abs(z) : norme de z, `z.norm()` : norme au carré !
- Initialisation : `std::complex<double> z(3,4); // 3 + 4i`
- Depuis C++ 14, possibilité écriture plus naturelle :

```
#include <iostream>
#include <complex>
using namespace std::complex_literals;

int main ( )
{
    std::complex<double> c = 1.0 + 1i ;
    std::cout << ”abs ” << c << ” = ” << std::abs(c) << ’\n’;
    std::complex<float> z = 3.0f + 4.0if;
    std::cout << ”abs ” << z << ” = ” << std::abs(z) << ’\n’;
    return EXIT_SUCCESS;
}
```

Les caractères et les chaînes de caractère

- Plusieurs représentations possibles pour les caractères :
  - **ASCII** : 128 caractères dont les lettres anglo-saxonnes codés sur sept bits + un bit de contrôle;
  - **UTF-8** : Tous les caractères connus codés sur un à quatre octets (non fixe)
  - **UTF-16** : Tous les caractères connus codés sur deux ou quatre octets (non fixe)
  - **UTF-32** : Tous les caractères connus codés sur quatre octets

```
char ascii = 'p';
char utf8 = u8'p'; // utf8 = u8'é' va générer une erreur car le caractère 'é ' > un octet
wchar_t utf16   = L'é' ;
char32_t utf32 = u'é ';
```

Caractère ASCII et unicode

- Seul ASCII bien supporté pour la gestion de caractère en C++
- Bibliothèque très pauvre pour les autres encodage;
- L'affichage correct sur console, ormis l'ASCII, non assuré : dépend du type d'encodage des caractères de la console…
- Attention : affichage différent d'encodage unicode du code source.

Chaînes de caractères natifs

- Même type qu'en C Pour ASCII et unicode :

"Ceci est une chaîne de caractère !"
u8" π est un caractère très spécial !"

- **Peut être stocké dans un char*…**
- **char16_t* pour utf_16, char32_t* pour utf32…**

```
char *texte = u8" π est un caractère très spécial !";
```

Chaîne de caractères brutes

- Chaîne de caractères sans interprétation des caractères
- N'interprète pas un retour à la ligne, un double quote, etc.
- Défini par un R avant le début et des délimiteurs

```
char * raw_text = R"RAW(Ceci est une "chaîne"
        où on peut retourner à la ligne
        où encore mettre un ")RAW";
std::cout << raw_text << std::endl;
```

```
Ceci est une "chaîne"
        où on peut retourner à la ligne
        où encore mettre un "
```

Chaîne de caractères std::string

- Utiliser la bibliothèque <string>
- Permet une manipulation plus aisée des chaînes de caractères;
- Allocations et deallocations automatiques
- De nombreux services de proposés

```
std::string chaine = "ceci est une chaine";
std::string chaine2 = " est une autre chaîne";
std::string chaine3 = chaine + " ou " + chaine2;
int lgth = chaine3.length();
int pos = chaine3.find("une");
int pos2 = chaine3.find("une", pos+1);
std::string foundstr = "Occurrences de une à " +
                       std::to_string(pos) + " et " +
                       std::to_string(pos2) + ". ";
```

Initialisation directe d'une sd::string

- Initialiser une std::string par une chaîne de caractères native pas optimale : copie de la chaîne native.
- Depuis C++14, possibilité de définir directement une chaîne entre double quote comme une std::string
- Rajout d'un s après le dernier double quote

```
#include <string>
using namespace std::string_literals;// Indispensable
int main()
{
  std::string troisième_chaîne =
                                u8"Ceci est une std::string"s ;
}
```

Déclaration automatique implicite et explicite

Lorsqu'on déclare et initialise une variable, il y a redondance du type de la variable

```
int i = 4; // i déclaré entier, et initialisé avec entier
double x = 4.; // x double, initialisé avec un double
int j = 3.5;// Initialisation bizarre… Un bogue ?
int d = x/3; //Idem, vraiment voulu par le programmeur ?
```

- En Python, le type d'une variable est définie par la valeur qu'elle contient.
- En C++, il est possible de définir une variable dont le type dépendra du type de la valeur qui l'initialise : c'est une déclaration implicite
- On peut également déclarer une variable dont le type dépendra d'une expression explicite (mais qui ne sert pas à l'initialiser) : c'est une déclaration explicite

Déclaration automatique implicite d'une variable

- On utilise le mot clef natif `auto`

```
auto i = 4; // i de type int
auto x = 3.; // x de type double
auto z = 3.+4.i; // z de type complex<double>
auto nz = std::abs(z);// nz de type double
auto j; //Erreur compilation, impossible déduire type de j
i = 4.3; //i vaut maintenant 4 puisqu'il a été déclaré int
```

La déclaration implicite peut simplifier le code
Ne pas en abuser, sous peine de rendre le code peu lisible :

```
auto x = initialisation_echantillon(); //Type de x ???
auto y = 2*x/3; // Division entière, réelle ?
auto z =std::abs(y); //z même type que y ?
```

Déclaration automatique explicite d'une variable

Type de variable déduite à partir d'une expression

```
int i = 3;
decltype(i) j = 3*i;          // j est un int
decltype(4.*i) x = 4.*i+2.; // x est un double
```

Peu utile à ce niveau, mais on y reviendra où la déclaration automatique explicite (ou implicite) est indispensable !

Renommage de type

- typedef toujours valable dans les cas simples;
- On peut également utiliser à partir de C++ 11 le mot clef using

```
typedef double reel;
using reel = double;
```

Nous verrons que dans certains cas, typedef n'est pas utilisable et using indispensable.

On n'utilisera plus désormais que le mot clef using pour le renommage de type.

Initialisation des variables

- Plusieurs façons d'initialiser les variables en C++
  - À la C : `int i = 3;`
  - À la C++ 98 : `int i(3); // Par construction`
  - À la C++ 11 : `int i{3}; // Par liste d'initialisation`
- L'initialisation par construction nécessaire pour les variables nécessitant plusieurs paramètres.

```
std::string chaîne = "Bonjour le monde !"s;
std::string sous_chaîne(chaîne, 11, 5); // Vaut "monde"
```

- La notion de liste d'initialisation en C++ 11 est très importante. Elle permet d'initialiser une collection de valeurs. Elle existait déjà en C, mais généralisée en C++ 11 :

```
double vect3D[] = { 1., 3., 5.};
```

Autres possibilités pour l'initialisation

- Ecriture en binaire possible dès C++ 11

```
int xb = 0b0011000111001;
```

- Possibilité de mettre des séparateurs dans un nombre pour une écriture plus claire de ce nombre (C++ 14)

```
std::int32_t xb = 0b0'0110'0011'1001;
std::int64_t value = 1'350'450'000LL;
double pi = 3.14'15'92'65'36;
```

Structures en C++

- Plus besoin de typedef
- Initialisation des structures facile avec les listes d'initialisation
- À partir de C++ 20, possibilité d'initialisation partielle en désignant les champs initialisés (avec g++, possible dès C++ 11, mais pas dans la norme)
- Possibilité de définir une structure dans une fonction

```
struct fiche_élève
{
    std::string  prénom, nom;
    std::int32_t âge, numéro_étudiant, promotion;
};
fiche_élève fiche1; // fiche non initialisée
fiche_élève fiche2{"Henry"s, "Chambier"s, 33, 113293, 2022};
fiche_élève fiche3{.prénom="Paul"s, .nom="Pierre"s,
                   .promotion=2022};//initialisation partielle
```

Le qualificateur const

- Certaines variables ne doivent pas changer de valeur. Par exemple : π
- Pour empêcher cela, on utilise le mot clef `const`
- C'est un qualificateur : il se met avant ou après le type de la variable.

```
const double π = 3.141592653589793;
double const e = 2.718281828459045;

π = 3.; // Erreur de compilation !
```

Pointeurs natifs en C++

- Même chose qu'en C
- Seul changement notable : pour le pointeur nul, on utilise `nullptr`
- `nullptr` est de type `std::nullptr_t`. On verra l'intérêt pour les fonctions.
- Attention à la signification de `const` pour les pointeurs :

```
int i = 3, j = 4;
int const* pt_i = &i; // Pointeur considérant i comme const
*pt_i = 3; // Erreur compilation
pt_i = &j; // OK
int * const pt_j = &i; // Pointeur const sur i
*pt_j = 3; // OK
pt_j = &j; // Erreur compilation
int const* const pt_k= &i; // Pointeur const sur i const
*pt_k = 3; // Erreur compilation
pt_k = &j; // Erreur compilation
```

Pointeurs partagés

- Pointeur comptant le nombre de pointeurs se référant à sa valeur;
- La valeur n'est détruite que lorsque le dernier pointeur s'y référant est détruit;
- Assure de ne pas avoir de fuite mémoire;
- La valeur est initialisée en même temps que le pointeur.

```
#include <memory>

std::shared_ptr<int> pt_int = nullptr;
pt_int = std::make_shared<int>(4);
auto pt_double = std::make_shared<double>(3.14);
auto pt_fiche = std::make_shared<ficheEtudiant>("Robert"s,
                                  "Chambier"s, 33, 152743, 2022);
```

Pointeurs partagés (suite)

- Impossible de faire une initialisation partielle de structure (obligatoirement complète ou aucune)
- Les pointeurs se détruisent comme toute variable : à la sortie de leur bloc d'instruction
- Gestion des pointeurs un peu plus lente que pour les pointeurs natifs (compteur de référence)
- Se manipule comme les pointeurs natifs (sauf arithmétique)
- On peut accéder au pointeur natif sous-jacent

```
auto pt_i = std::make_shared<int>(4);
auto pt_j = std::make_shared<int const>(5);
*pt_i = 3;
*pt_j = 4;// Erreur compilation
auto pt_fiche = std::make_shared<ficheEtudiant>();
pt_fiche->prénom = "Robert"s;
```

Pointeurs partagés (suite…)

```
auto pt_i = std::make_shared<int>(4);
std::cout << "Nbre réf. sur 4 : " << pt_i.count() << std::endl;
{
  auto pt_j = pt_i;
  std::cout << "Nbre réf. sur 4 : " << pt_i.count() << std::endl;
  {
    auto pt_k = pt_i;
    std::cout << "Nbre réf. sur 4 : " << pt_i.count() << std::endl;
  }
  std::cout << "Nbre réf. sur 4 : " << pt_i.count() << std::endl;
}
std::cout << "Nbre réf. sur 4 : " << pt_i.count() << std::endl;
int* natif_pt = pt_i.get();
```

```
Nbre réf. sur 4 : 1
Nbre réf. sur 4 : 2
Nbre réf. sur 4 : 3
Nbre réf. sur 4 : 2
Nbre réf. sur 4 : 1
```

Pointeurs uniques

- Un seul pointeur à la fois peut pointer sur une valeur crée par un pointeur unique;
- La valeur se détruit quand le dernier pointeur se référant à cette valeur est détruit;
- Gestion aussi rapide que les pointeurs natifs sans fuite mémoire
- Se manipule comme les pointeurs natifs (sauf arithmétique)
- On peut accéder au pointeur sous-jacent

```
#include <memory>

std::unique_ptr<double> pt_d = nullptr;
auto pt_z = std::make_unique<std::complex<double>>(-1.,3.1415);
auto pt_f = std::make_unique<ficheEtudiant>("Robert"s,
                                            "Chambier"s, 33, 152743, 2022);
auto pt_z2 = pt_z; // Erreur compilation
auto pt_z3 = std::move(pt_z); //OK, déplacement
```

Copie contre déplacement

- En C++, à partir du 11, on peut copier ou déplacer des valeurs
- `int c = d;` ➔ Copie, c possède la même valeur de d
- `int c = std::move(d);` ➔ Déplacement, c vole la valeur à d qui ne possède plus sa valeur après cet appel
- Pour les pointeurs uniques, seul le déplacement est possible

```
auto pt_fiche1 = std::make_unique<ficheEtudiant>("Robert"s,
                                                 "Chambier"s, 33, 152743, 2022);
std::cout << "pt_fiche1 en : " << (void*)pt_fiche1.get() << std::endl;
auto pt_fiche2 = std::move(pt_fiche1);
std::cout << "pt_fiche1 en : " << (void*)pt_fiche1.get() << std::endl;
std::cout << "pt_fiche2 en : " << (void*)pt_fiche2.get() << std::endl;
```

```
pt_fiche1 en : 0x696b80
pt_fiche1 en : 0
pt_fiche2 en : 0x696b80
```

Les références

- Ne stocke pas de valeur, mais fait référence à une valeur existante en mémoire;
- Si la valeur est modifiée, la référence verra la valeur modifiée;
- Si on fait référence à une valeur possédée par une variable, on peut voir cela comme un alias à cette variable;
- Une référence doit obligatoirement faire référence à une valeur stockée en mémoire;
- On rajoute le symbole & pour déclarer une référence.

```
int i = 3, j = 4;
int& k = i; // k faire référence à la valeur stockée par i
k = 2; // Maintenant i et k valent deux !
i = 1; // Maintenant i et k valent un !
k = j; // Maintenant i et k valent 4…
```

Les références (suite)

- Une référence n'est pas obligée de faire référence à une valeur stockée dans une variable

```
auto pt_x = std::make_unique<double>(-0.707);
double& x = *pt_x;
x = 0.707; // la variable pointée par pt_x vaut 0.707
*pt_x = 1.414; // x voit la valeur 1.414
auto pt_y = std::move(pt_x);
x = 3.1415; // pt_y pointe sur la valeur 3.1415
*pt_y = 2.28; // x voit maintenant la valeur 2.28
```

- On peut déclarer une référence sur une valeur considérée comme `const` mais cela n'empêche pas de modifier la valeur par un autre moyen !

```
int i = 4;
int const& j = i;
j = 4; // Erreur de compilation !
i = -11; // OK, j voit maintenant -11 comme valeur !
```

Gestion statique et dynamique de la mémoire

- **Allocation statique** : On connaît durant la compilation la taille à réserver : le compilateur réserve dans l'espace de l'exécutable un espace pour stocker les données
- **Allocation dynamique** : On ne connaît pas à la compilation la place mémoire à réserver : c'est durant l'exécution du programme qu'on réserve la mémoire
- Allocation statique : sur la pile
- Allocation dynamique : sur le tas
- Pile limitée par la taille sur certains systèmes d'exploitations (Windows… Entre 512ko et 2Go)
- Ne pas allouer de grande taille en statique !

Allocation dynamique de variables

- Allocation statique variable = déclaration variable
- Exemple allocation dynamique : liste simplement chaînée
- Pour réserver une valeur en mémoire : opérateur **new**

```
struct liste_entier
{
    int valeur;
    liste_entier* prochain;
};
…
// Construction
liste_entier racine{1,nullptr}; // Initialisation de la racine
racine.prochain = new liste_entier(2,nullptr);
liste_entier* nœud = racine->prochain;
nœud->prochain = new liste_entier(3,nullptr);
nœud = nœud->prochain;
…
```

Allocation dynamique de variables

- Pour désallouer une valeur en mémoire : opérateur **`delete`**

```
// Destruction de la liste
liste_entier* prochain = root.prochain;
delete prochain->prochain->prochain;
delete prochain->prochain;
delete prochain;
```

- Attention, ne jamais mélanger malloc/free avec new/delete !
- Ne pas oublier de faire un **`delete`** pour chaque **`new`** d'appeler
- Sinon on aura une fuite mémoire;
- L'allocation dynamique avec **`new`**/**`delete`** indispensable jusqu'à C++ 11
- Beaucoup moins utile depuis C++ 11 ➔ **Il vaut mieux éviter le plus possible de les utiliser.**

Allocation dynamique de variables

- On peut allouer des variables dynamiquement avec les pointeurs partagés ou uniques !
- Bien plus sûr et impossible d'avoir des fuites mémoires
- Exemple pour la liste :

```
struct liste_entier
{
  int valeur;
  std::unique_ptr<int> prochain; // Ou std::shared_ptr
};

// Construction
liste_entier racine(2,nullptr);
racine.prochain = std::make_unique<liste_entier>(3,nullptr);
auto& prochain = racine.prochain;
prochain->prochain=std::make_unique<liste_entier>(4,nullptr);
auto& prochain2 = prochain->prochain;
…
```

Allocation dynamique de variables

- La destruction bien plus facile à écrire :

```
racine.prochain = nullptr;
```

- Suppression d'un nœud

```
std::unique_ptr<liste_entier>& nœud_précédent = …
nœud_précédent->prochain = nœud_précédent->prochain->prochain;
```

Exercice :

1. Créer une liste contenant des entiers pour valeurs et pointant sur le suivant à l'aide de pointeurs partagés
2. Déclarer et définir des fonctions qui permettent : d'initialiser la racine de la liste, rajouter une valeur à la fin de la liste, de supprimer tous les multiples d'une valeur dans la liste (la valeur exclue)
3. Créer dans le programme principal une liste contenant la valeur deux et les valeurs impaires supérieures à deux et inférieurs à un certain N fixé.
4. À l'aide du crible d'Eratosthène, ne conserver que les nombres premiers dans la liste et les afficher à l'écran

Les conteneurs en C++

Qu’est ce qu’un conteneur ?

- Un conteneur est un type de valeur qui contient une collection d’autres valeurs
- Un conteneur gère de lui-même la réservation et la libération de la mémoire
- On peut accéder en lecture ou en écriture aux valeurs d’un conteneur
- Dans les conteneurs, on a de proposé en C++ :
  - Les tableaux statiques
  - Les tableaux dynamiques
  - Les listes
  - Les queues, les tas et les piles
  - Les arbres, les tableaux associatifs (les dictionnaires)
  - Etc.
- Tous les conteneurs possèdent des itérateurs
- Mais c’est quoi un itérateur ?

Les itérateurs en C++

- Un itérateur est un type de variable qui pointera sur des valeurs d'un conteneur (un tableau, une liste, etc.)
- Il possède la faculté de pouvoir parcourir les valeurs d'un conteneur à l'aide d'opérateurs dont ++ et *
- L'itérateur le plus simple, qui existe en C est le pointeur

```
double tableau[] = {1,2,3,4,5,6,7,8,9,10};
double *pt_coef = tableau;
for ( int i = 0; i < 10; ++i )
{
  std::cout << *pt_coef << " ";
  ++ pt_coef;
}
```

- L'itérateur d'une liste sera plus complexe !

Les itérateurs en C++ (suite)

- Plusieurs types d'itérateurs :
  - Les itérateurs de lecture ou/et écriture. Exemple : itérateur sur des valeurs constantes ou non
  - **Les itérateurs uni-directionnel** : on ne peut qu'avancer vers le prochain élément. Exemple : itérateur d'une liste simplement chaînée
  - **Les itérateurs bi-directionnels** : on peut aller vers le prochain élément ou le précédent. Exemple : itérateur d'une liste doublement chaînée
  - **Les itérateurs à accès aléatoires** : on peut avancer ou reculer, sauter des éléments, etc. Exemple : itérateur d'un tableau
- La fonction `begin` créee un itérateur sur le 1er élément d'un conteneur et la fonction `end` un itérateur sur la fin du conteneur

```
std::list<double> liste; //Spoil ! On verra la liste plus loin
for ( auto iter_beg = liste.begin(); iter_beg != liste.end();
      ++iter_beg)
{
   double& x = *iter_beg;
   *iter_beg = 3;// Si la liste n'est pas constante !
}
```

Les tableaux statiques

- Tableau statique réserver à la compilation
- Possibilité de déclarer des tableaux statiques à la C
- Mais pas de contrôle possible des indices…
- En C++, bibliothèque <array> propose tableau statique
- Contrôle initialisation et accès aux éléments si besoin

```
#include <array>
double vecteur3D[] = {1.,0.,0.}; // À la C

std::array<double,3> vecteur3D = {1.,0.,0.};// À la C++
std::array<double,3> e1 = {1.,0.,0.,0.};// Erreur compilation !
std::array<double,3> e2 = {0.,1.}; // Equivalent à {0.,1.,0.}
std::array<double,3> e3;// Non initialisé
std::array<std::array<double,3>,3> A = {
                        std::array{1.,2.,3.}, // Syntaxe C++ 17
                                   {2.,3.,1.},
                                   {3.,1.,2.}
                                    };
```

Les tableaux statiques

- Depuis C++ 17, on peut simplifier la syntaxe

```
std::array tableau ={1.,2.,3.,4.};// Tableau de quatre doubles
```

- Le C++ « devine » le type d'éléments et la longueur du tableau statique
- On ne contrôle plus la taille du tableau avec ce type d'initialisation : gare au bogues
- Mais allège considérablement l'écriture du code (donc plus facile à lire !)
- De toute façon, on peu continuer à contrôler l'accès aux données :

```
double x = tableau[0];//ok
// Erreur à l'exécution avec gcc si l'option -D_GLIBCXX_DEBUG
// a été mis en option de compilation
double y = tableau[4];
double z = tableau.at(4);// Erreur levée systématiquement
double* tab = tableau.data();// tab pointe sur début tableau
double& rx = tableau[2];
rx = 4.; // On modifie le troisième élément de tableau
double& début = tableau.front(), fin = tableau.back();
```

Les tableaux statiques

```
auto size = tableau.size();//Nombre d'éléments dans le tableau
tableau.fill(-1.);// Remplit le tableau avec des -1.
std::array<double,4> buffer;
buffer.swap(tableau);// Permute les données de tableau avec buffer
```

- La fonction swap a un coût linéaire en fonction de la taille du tableau. On permute les éléments un par un
- Il est possible de comparer lexicographiquement deux tableaux, à condition qu'ils aient le même type d'éléments et qu'ils soient comparables

```
std::array parties_entières = {-1., -2., 3., 6., -4.};
std::array valeurs_réelles  = {-1.2, -2.3, 3.4, -6.5, -4.1};
if ( parties_entières > valeurs_réelles )
    std::cout << "La partie entière possède des éléments plus gros !";
```

```
La partie entière possède des éléments plus gros !
```

Parcours des tableaux en C++

- Un conteneur est une collection de valeurs : tableau statique, dynamique, liste, arbre, dictionnaire, etc.
- Plusieurs façons de parcourir un conteneur :
  - À la C avec une boucle for classique
  - En itérant sur les valeurs du tableau par référence ou copie
  - En itérant explicitement avec un itérateur

```
for (decltype(tableau.size()) i = 0; i < tableau.size(); ++i ) // Boucle classique en C
{
   double& rx = tableau[i];
   …
   rx = 4.; …
}

for ( auto& réf : tableau ) // C++ 11 et supérieur : boucle à la Python…
{  …   réf = 4.;  …  }

for ( auto iter = tableau.begin(); iter != tableau.end(); ++iter) // Par itérateur
{ … }
```

Exercice sur les tableaux statiques

- Calcul de l'aire signée
  - Définir le type point et le type vecteur comme des tableaux à deux coefficients double précision
  - Définir une structure triangle comme un tableau de trois points p_1, p_2 et p_3.

Définir un point p{0,0} et un triangle T{p_1{-1,-1}, p_2{1,-1} et p_3{0,1}}

Calculer les vecteurs pp_1, pp_2 et pp_3.

Calculer le produit croisé (u x v = u.x*v.y – u.y*v.x) des trois vecteurs avec p, deux à deux (trois produits correspondant aux aires signés des trois triangles pp_1p_2, pp_2p_3, pp_3p_1)

- Vérifier que les trois scalaires obtenus ont le même signe
- Si oui, afficher que le point p est bien dans le triangle.

Tableaux dynamiques natifs

- Permet d'allouer un tableau à l'exécution;
- On peut utiles opérateurs `new[]` et `delete[];`
- **`new[]`** permet d'allouer mais pas d'initialiser.

```
int n = 20;
// Allocation d'un simple tableau d'indices
int *indices = new int[n];
// Allocation d'un tableau de coordonnées en 3D
auto coords = new std::array<double,3>[n];
// Allocation d'une matrice carrée de dimension n
double** matrice = new double*[n];
for (int i = 0; i < n; ++i) matrice[i] = new double[n];
```

- Initialisation semblable au C

```
for (int i=0; i<n; ++i) indices[i] = i+1;
for (int i=0; i<n; ++i) coords[i] = {1.5*i,2.5*i-
2.,2.1*i+4.};
for (int i = 0; i<n; ++i )
    for (int j = 0; j<n; ++j)
       matrice[i][j] = (i+j)%n+1;
```

Tableaux dynamiques natifs…

- Utilisation de `delete[]` pour désallouer

```
delete [] indices;
delete [] coords;
for (int i = 0; i<n; ++i) delete [] matrice[i];
delete [] matrice;
```

- Allocation/Désallocation très proche du C
- New plus facile à utiliser que malloc
- New et delete font parti du langage de base !
- Potentiellement sujets aux mêmes bogues !
  - Dépassement d'indice non contrôlé
  - Fuite mémoire assez courant avec ce type de code
- Pas de services proposés pour faciliter la gestion des tableaux

Tableaux dynamiques partagés ou uniques

- Utilisation de **`std::shared_ptr`** ou **`std::unique_ptr`** depuis C++ 17
- Création d'un tableau partagé en C++ 17

```
int n = 20;
std::shared_ptr<int[]> indices(new int[n]);
std::shared_ptr<std::array<double,3>[]> coords(new std::array<double,3>[n]);
std::shared_ptr<std::shared_ptr<double[]>[]>
                                    matrice(new std::shared_ptr<double[]>[n]);
for (int i = 0; i<n; ++i)
    matrice[i] = std::shared_ptr<double[]>(new double[n]);
```

- Création d'un tableau partagé en C++ 20

```
int n = 20;
auto indices = std::make_shared<int[]>(n);
auto coords  = std::make_shared<std::array<double,3>[]>(n);
// Pour la matrice, échec d'utilisation de cette approche(bogues compilateur ?)
```

- Même approche avec **`std::unique_ptr`**
- Accès identiques à ceux des tableaux dynamiques natifs.

Tableaux dynamiques partagés ou uniques…

- Pas de désallocation à faire. Cela se fait automatiquement !
- Même principe d'allocation qu'avec new (qu'on utilise en C++ 17 !)
- Mieux qu'une simple allocation/désallocation avec new/delete
  - Simplifie le code en enlevant la phase de désallocation
  - On est assurer de ne pas avoir de fuite mémoire
- Cependant :
  - Pas de contrôles possibles sur les indices d'accès aux valeurs
  - Pas de services proposés pour faciliter la vie du programmeur !

Tableaux dynamique avec vector

- Il existe un type tableau dynamique en C++
- Pas natif. Proposé dans la bibliothèque <vector>
- Puissant avec des stratégies d'allocation élaborées

```
#include <vector>
std::vector e1{1.,0.,0.};
int n = 20;
std::vector<int> indices(n, -1);// Avec initialisation à -1
std::vector<std::array<double,3>> coords(n);
std::vector<std::vector<double>> matrice;
matrice.reserve(n);
for (int i = 0; i<n; ++i)
    matrice.emplace_back(n);
```

- Possibilité d'une « liste de compréhension »

```
#include <algorithm>
std::generate(indices.begin(),indices.end(),[n=0]()mutable
{n+=1;return n;});
std::generate(coords.begin(),coords.end(),
              [n=-1](){n+=1;return {1.5*i,2.5*i-2.,2.1*i+4.}; });
```

Stratégie d'allocation de vector

- Deux notions importantes dans std::vector : la capacité et la taille
  - **La capacité** : Taille de la mémoire réservée (nombre d'éléments qu'on peut contenir avec la mémoire réservée)
  - **La taille** : Nombre d'éléments dans le vecteur. La taille est toujours plus petite que la capacité !

```
std::vector<double> u(30);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
u.resize(20);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
u.resize(40);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
std::vector<double>(30).swap(u);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
std::vector<double> v; v.reserve(100);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
v.push_back(3.14); v.push_back(2.28); v.push_back(1.);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
v.pop_back();
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
v.shrink_to_fit();
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
```

Stratégie d'allocation de vector

```
std::vector<double> u(30);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
u.resize(20);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
u.resize(40);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
std::vector<double>(30).swap(u);
std::cout<<"u => Capacité: "<<u.capacity()<<"\tTaille: "<<u.size()<<std::endl;
std::vector<double> v; v.reserve(100);
std::cout<<"v => Capacité: "<<v.capacity()<<"\tTaille: "<<v.size()<<std::endl;
v.push_back(3.14); v.push_back(2.28); v.push_back(1.);
std::cout<<"v => Capacité: "<<v.capacity()<<"\tTaille: "<<v.size()<<std::endl;
v.pop_back();
std::cout<<"v => Capacité: "<<v.capacity()<<"\tTaille: "<<v.size()<<std::endl;
v.shrink_to_fit();
std::cout<<"v => Capacité: "<<v.capacity()<<"\tTaille: "<<v.size()<<std::endl;
```

```
u => Capacité : 30      Taille : 30
u => Capacité : 30      Taille : 20
u => Capacité : 40      Taille : 40
u => Capacité : 30      Taille : 30
v => Capacité : 100     Taille : 0
v => Capacité : 100     Taille : 3
v => Capacité : 100     Taille : 2
v => Capacité : 2       Taille : 2
```

Services associés à vector

- Rajout/Suppression d'un élément à la fin :
  - Copie une nouvelle valeur à la fin : push_back
  - Initialise une nouvelle valeur à la fin : emplace_back
  - Eliminer le dernier élément du vecteur : pop_back
- Opérateurs d'accès :
  - Comme en C, avec les [ ], pas de contrôle normalement sauf si on rajoute `-D_GLIBCXX_DEBUG` avec gcc;
  - Avec le service get : contrôle systématique des indices, plus lent que d'accéder avec les [ ]
  - Pointeur sur le début du tableau : data()
  - Itérateurs avec begin() et end(), rbegin() et rend(), etc.
  - Et d'autres encore (back, front, etc.)
- Copie, déplacement, échange et comparaisons

Exemple d'utilisation de vector

```
int N = 100;
[[maybe_unused]] int i;
std::vector<std::int32_t> carrés(N);
std::generate(carrés.begin(),carrés.end(),[i=0]() mutable { i++; return i*i; });
std::vector<std::int32_t> non_pythagoricien; non_pythagoricien.reserve(100);
for (auto iter_1=carrés.begin(); iter_1!=carrés.end(); ++iter_1)
{
   bool est_somme = false;
   for(auto iter_2=carrés.begin();(iter_2!=iter_1)&&(not est_somme);++iter_2)
   {
     for(auto iter_3=carrés.begin();(iter_3!=iter_2)&&(not est_somme);++iter_3)
     {
       if ((*iter_2)+(*iter_3) == *iter_1) est_somme = true;
     }
   }
   if (not est_somme) non_pythagoricien.emplace_back(*iter_1);
}
non_pythagoricien.shrink_to_fit();
std::cout << "Nombre de carrés non pythagoryciens : " << non_pythagoricien.size()
          << std::endl;
for (auto val : non_pythagoricien) std::cout << val << " ";
std::cout << std::endl;
```

Recherche de carrés d'entiers qui ne sont pas la somme de deux carrés entiers (nombre pythagoricien)

Exercice sur vector

- Crible d'Ératosthène
- Créer un vecteur contenant tous les entiers de deux à N (on pourra modifier N)
- On élimine les multiples en les mettant à zéro
- On parcourt le tableau pour rajouter dans un autre tableau (qui contiendra les nombres premiers) les entiers non nuls
- On affiche le nombre de nombres premiers trouvés et la liste de ces nombres

Les listes

- Les listes : bibliothèque <list>
  - Rajout à la fin, au début, suppression au milieu : O(1)
  - Itérateurs disponibles (begin, end, cbegin, cend,…)
  - Opérateurs de copie,déplacement, comparaison
  - Enlever des valeurs selon un critère, etc.
  - En général plus lent que les vecteurs
  - Mais utile si beaucoup d'insertion au début ou au milieu, etc.

```
std::list l1{5, 7, 11, 13, 17, 19, 23, 29};
std::list<std::int32_t> l2{2,3};
for (int i = 1; i < 100; ++i ) {
  l2.emplace_back(6*i-1);
  l2.emplace_back(6*i+1);
}
for ( auto val : l1 )
  l2.remove_if([val](int n) { return (n>val) && (n%val == 0); });
std::cout << "Nombres premiers (" << l2.size() << ") : ";
for ( auto val : l2 ) std::cout << val << " ";
std::cout << std::endl;
```

Exercice sur les listes

- Reprendre la structure fiche d'étudiant
- Créer une liste contenant plusieurs étudiants dont certains de la même promotion
- Afficher la liste
- Trier la liste par nom (voir le service sort de list sur Cppreference)
- Supprimer les étudiants d'une promotion donnée
- Afficher le nombre d'étudiants contenus dans la liste

Les dictionnaires

- Deux dictionnaires : `std::map` et `std::unordered_map`
  - `std::map` : dictionnaire trié
  - `std::unordered_map` : dictionnaire avec table hashage

```
#include <map>
std::map<std::string, int> m{ {"CPU", 10},{"GPU", 15},{"RAM", 20},};
for (const auto& [key, value] : m)
{
    std::cout << key << " = " << value << "; ";
}
std::cout << std::endl;
m["CPU"] = 25; // Màj d'une valeur existante
m["SSD"] = 30; // Création d'une nouvelle valeur
```

```
#include <unordered_map>

std::unordered_map<std::string, std::string> u =
    { {"ROUGE","#FF0000"}, {"VERT","#00FF00"}, {"BLEU","#0000FF"} };
for( const auto& [key, value] : u ) {
    std::cout << "Clef:[" << key << "] Valeur:[" << value << "]\n"; }
u["NOIR"] = "#000000"; u["BLANC"] = "#FFFFFF";
std::cout << "La valeur HEX de ROUGE est  :[" << u["RED"] << "]\n";
std::cout << "La valeur HEX de NOIR est :[" << u["BLACK"] << "]\n";
```

Exercice sur les dictionnaires

- Reprendre la structure ficheEtudiant
- Créer un dictionnaire dont la clef est le nom et la valeur la fiche
- Essayer diverses manipulation avec map et unordered map…
- N'hésitez pas à aller regarder ce qu'on peut faire avec sur CppReference

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

Les fonctions en C++

- Syntaxes de déclaration des fonctions
- Déduction automatique du type de la valeur de retour
- Retour de valeurs multiples
- Retour par référence
- Passage par référence et référence universelle
- Surcharge des fonctions
- Valeurs par défauts
- Fonction générique (C++20)
- Surcharge des opérateurs

Syntaxe de déclaration

- Plusieurs façons de déclarer une fonction
  - À la manière du C
  - À la manière de la programmation fonctionnelle

```
double dot_c(std::int64_t n, const double* x, const double* y)
{
    double sum = 0;
    for ( std::int64_t i = 0; i < n; ++i )
        sum += x[i]*y[i];
    return sum;
}

auto dot_f1(std::int64_t n, const double* x, const double* y) -> double
{
    double sum = 0;
    for ( std::int64_t i = 0; i < n; ++i )
        sum += x[i]*y[i];
    return sum;
}
```

Déduction automatique du type de la valeur de retour

- Deux possibilités pour la déduction automatique du retour
  - On déclare le type de la valeur retournée ➔ On peut retourner une simple liste d'initialisation si la valeur retournée à besoin de plusieurs valeurs
  - On déclare le type de la valeur retournée comme auto : le type de la valeur retournée sera déduite de la mise en œuvre
    - Attention, dans le cas où il y a plusieurs return dans la mise en œuvre de la fonction, il ne faut pas retourner plusieurs types différents sous peine d'erreur de compilation

Déduction automatique du type de la valeur de retour…

```
auto produit_conjugué( std::complex<double> z1,
                       std::complex<double> z2 )
{
    return std::complex{z1.real()*z2.real()+z1.imag()*z2.imag(),
                        z2.real()*z1.imag()-z1.real()*z2.imag()};
}

auto produit_conjugué ( std::complex<double> z1,
                        std::complex<double> z2 ) -> decltype(z1)
{
    return {z1.real()*z2.real()+z1.imag()*z2.imag(),
            z2.real()*z1.imag()-z1.real()*z2.imag()};
}
```

Retour de valeurs multiples

- En C, on retourne une valeur, les autres passés en pointeur à la fonction
  - Si un pointeur est nul, que fait-on ?
  - L'argument passé en pointeur sert-il aussi en argument d'entrée ?
  - Lourdeur d'écriture : les arguments passés en pointeurs doivent être déférencés dans la mise en œuvre.
  - On doit passer ces arguments par adresse : alourdit également l'écriture

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

Retour valeurs multiples…

- En C++, on peut renvoyer un tableau de deux entiers

```
std::array<int,2> division_euclidienne(int p, int q)
{
    int résultat = p/q;
    return {résultat, p - résultat*q};
}
auto res = division_euclidienne(p,q);// res[0]=résultat, res[1]=reste
```

  - Plus de problème de pointeur nul, et séparation des valeurs en entrée et des valeurs en sortie
  - Mais tableau peu expressif pour savoir l'ordre des résultats
  - Depuis C++ 17, possibilité de recevoir les valeurs de certaines structures statiques dans des variables multiples :

```
std::array<int,2> division_euclidienne(int p, int q)
{
    int résultat = p/q;
    return {résultat, p - résultat*q};
}
auto [résultat, reste] = division_euclidienne(p,q);
```

Retour valeurs multiples…

- Fonctionne pour retourner plus de deux valeurs, mais de types homogènes :

```
std::array<std::complex<double>,3>
racines_cubique(std::complex<double> z)
{
    const double π_2s3 = 2*std::numbers::pi/3.;
    const double π_4s3 = 4*std::numbers::pi/3.;

    double argument = std::arg(z)/3.;
    double module   = std::cbrt(std::abs(z));//cbrt = Racine cubique

    return {
        module * std::exp(1.i* argument),
        module * std::exp(1.i*(argument + π_2s3)),
        module * std::exp(1.i*(argument + π_4s3))
    };
}
…
 auto [z1,z2,z3] = racines_cubique(1.+1.i);
```

Retour valeurs multiples…

- Comment faire si valeurs retournées de types hétérogènes ?
  - Si seulement deux valeurs à retournées : utiliser les paires de valeurs (`std::pair` dans bibliothèque `utility`)
  - Si plus de deux valeurs à retournées : utiliser un `tuple` (bibliothèque `tuple`)
  - Pour une paire de valeurs :

```
std::pair<double,int>
trouve_et_localise_valeur_maximale( int n, const double * values)
{
    int index = 0;
    int value_max = values[0];
    for ( int i = 1; i < n; ++i )
    {
        if (value_max < values[i])
        {
            value_max = values[i];
            index     = i;
        }
    }
    return {value_max,index};
}
auto [valeur,index] = trouve_et_localise_valeur_maximale(n,tableau);
```

Retour valeurs multiples…

- Tuple : comme en Python, collection fixe de valeurs hétérogènes
- Assez compliqué à déclarer en C++ :
  - On utilise auto pour la déduction automatique du type de retour
  - On utilise `std::make_tuple` pour retourner le tuple

```
auto convertion_en_fraction(double valeur, int nombre_iter_max)
{   // Utilise un développement en fraction continue…
    std::int64_t dividende, diviseur;
    assert(valeur>=0.);
    std::int64_t partie_entière = std::int64_t(valeur);
    double reste = valeur - partie_entière;
    if (std::abs(reste)<1.E-14)
    return std::make_tuple(partie_entière, 1LL, 0.);
    if (nombre_iter_max == 0)
    return std::make_tuple(partie_entière, 1LL, reste);
    auto [p,q,r] = convertion_en_fraction(1./reste, nombre_iter_max-1);
    dividende = partie_entière*p+q;
    diviseur  = p;
    reste = std::abs(valeur-double(dividende)/double(diviseur));
    return std::make_tuple(dividende, diviseur, reste);
}
auto [diviseur,dividende,reste]=convertion_en_fraction(std::sqrt(2),20);
```

Retour valeurs multiples…

- On veut parfois ignorer une des valeurs retournées
  - Dans le fonction précédente, on n'a pas besoin d'avoir la valeur du reste

```
auto [p,q,r] = convertion_en_fraction(1./reste, nombre_iter_max-1);
```

- Deux façons d'ignorer ce retour (sans warning de la part du compilateur)
  - On récupère les trois valeurs en précisant que certaines ne seront pas utilisées
  - On récupère les trois valeurs avec la fonction std::tie qui permet de récuper les valeurs d'un tuple dans des variables préalablement déclarées en remplaçant la valeur à ignorer par **`std::ignore`**

```
[[maybe_unused]] auto [p,q,ignorée] =
                    convertion_en_fraction(1./reste, nombre_iter_max-1);
```

```
std::int64_t diviseur, dividende;
std::tie(diviseur,dividende,std::ignore)=
                    convertion_en_fraction(1./reste, nombre_iter_max-1);
```

Retour d'une variable locale/Retour d'une variable globale

- Retour variable locale à une fonction = déplacement
  - Retour d'un tableau dynamique, liste, etc: aucun coût mémoire et CPU (échange de pointeur);
- Retour d'une variable globale = copie de cette variable : éviter de la renvoyer en retour d'une fonction

```
std::vector<double>
add( const std::vector<double>& u, const std::vector<double>& v )
{
    assert(u.size() == v.size());
    std::vector<double> w(u.size());
    for ( int i = 0; i < u.size(); ++i )
        w[i] = u[i] + v[i];
    return w; // OK, on effectue un déplacement, aucun coût en mémoire
}

std::vector<double>
add_inplace( const std::vector<double>& u, std::vector<double>& v)
{
    assert(u.size() == v.size());
    for ( int i = 0; i < u.size(); ++i )
        v[i] += u[i];
    return v;// Mauvais : on retourne une copie de v
}
```

Retourner une référence

- Quand une variable est globale, on peut retourner une référence sur cette variable
  - Attention, pour éviter une copie, il faudra recevoir une référence

```
std::unique_ptr<opengl::context> contexte = nullptr;
// Design pattern : singleton (Design Patterns: Elements of Reusable Object-Oriented Software)
opengl::context& get_context()
{
  if (contexte == nullptr) contexte = std::make_unique<opengl::context>();
  return *contexte;
}
…
auto& gl_contexte = get_context();
```

- Ne jamais retourner une variable locale en référence
  - On retourne une référence sur une variable qui sera détruite juste après !

```
double& résout_équation_linéaire(double a, double b)
{ // Résout l'équation a.x + b = 0
  std::assert(a!= 0);// Précondition
  return -b/a; // Erreur, on retourne une variable locale en référence
}
```

Passage par référence

- On peut passer par référence des arguments
  - Lors de l'appel, l'argument devient une référence de la valeur passée en argument
  - Si on modifie la valeur référencée par l'argument, on modifie la valeur passée en argument
  - Plus sûr et plus léger que le passage par pointeur : on est garantit qu'on fait une référence sur une valeur existante !
  - Pour une référence non constante : la valeur doit déjà exister avant passage par argument (variable ou coefficient de tableau)
  - Pour une référence constante : le C++ construit temporairement une valeur qu'il passe en argument si on passe une valeur directement

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

Passage par référence…

- Non constante : permet modification valeur passée en argument ;
- constante : permet passage arguments volumineux en mémoire
- Non constante : impossible de passer directement une valeur ;
- Constante : possibilité de passer directement une valeur.

```
void axpy( double a,
           const std::vector<double>& u, std::vector<double>& v)
{
    std::assert(u.size() == v.size());
    for (std::int64_t i = 0 ; i < u.size() ; ++i)
        v[i] += a*u[i];
}
...
std::vector v{1.,2.,3.};
axpy(0.5, {1.,-1.,1.}, v);
```

Référence « rvalue »

- Parfois, on veut que dans certains cas, on ne peut utiliser que des valeurs « temporaires » en argument
  - Soit des valeurs passées directement ;
  - Soit en déplaçant des valeurs contenues dans des variables.
- On utilise une « double référence » ;
- Attention, on verra une autre signification dans le cadre des fonctions génériques de cette « double référence »

```
void set_data( std::vector<double>& u, std::vector<double>&& temp)
{
    temp.swap(u);
}
...
std::vector u{1.,2.,3.,4.};
set_data(u, {1.,2.,5.,6.,3.,7.});
std::vector v{-1.,-2.,-3.,-4.};
// set_data(u, v); // Erreur de compilation
```

Surcharges des fonctions

- Plusieurs fonctions peuvent avoir le même nom du moment que les paramètres diffèrent et ne laissent pas d'ambivalence

```
void axpy ( int N, float a, const float *x , float *y)
{// Op. y <- y + a.x sur vecteurs x, y avec a scalaire
    for (int i=0; i<N; ++i) y[i] += a*x[i];
}
// Version double précision
void axpy ( int N, double a, const double *x, double *y)
{// Op. y <- y + a.x sur vecteurs x ,y avec a scalaire
    for (int i = 0; i<N ; ++i ) y[i] += a*x[i];
}
void main (int args, char* argv[])
{
    float fx[] = {1.f, 2.f, 3.f, 4.f};
    float fy[] = {0.f,-1.f,-2.f,-3.f};
    axpy (4,2.f,fx,fy); // Appel la version float
    double dx[] = {1., 2., 3., 4.};
    double dy[] = {0.,-1.,-2.,-3.};
    axpy (4,2.,dx,dy);  // Appel la version double
    axpy (4,2.f,dx,dy); // Erreur compilation
```

Fonctions génériques (C++ 2020)

- Écrire la même fonction avec la même mise en œuvre plusieurs fois pour des types différents : pénible et source de bogue
- Utilisation du type auto en paramètre
- On verra plus tard les templates qui font la même chose avec plus de contrôle

```
// Fonction générique pour tout type de vecteur
void axpy ( int N,auto a,const auto *x, auto *y )
{ // Op. y <- y + a.x sur vecteurs x ,y avec a scalaire
  for ( int i = 0; i<N; ++i ) y[i] += a*x[i];
}
void main ( )
{
  float fx[] = {1.f, 2.f, 3.f, 4.f};
  float fy[] = {0.f,-1.f,-2.f,-3.f};
  axpy (4,2.f,fx,fy); // Appel avec simple précision

  double dx[] = {1.,2.,3.,4.}, dy[] = {0.,-1.,-2.,-3.};
  axpy (4,2.,dx,dy); // Appel avec double précision
  axpy (4,2.f,dx,dy); // Appel a simple préc.,dx & dy double
```

Fonctions génériques (C++ 2020) (suite)

- A chaque nouveau jeu de paramètres (types), le C++ génère une nouvelle fonction
- Dans l'exemple précédent, trois fonctions ont été générées
- Le C++ ne générera une erreur que si l'opération `y[i] += a*x[i]` est incompatible avec les types donnés

```
axpy(4,2,fx,dy);// version (int, int, const float*,double*)
axpy(4,2,"toto","titi"); // ne compile pas
```

Première version compile : sens multiplication entier x réel
Deuxième version compile pas : multiplication entier par chaîne caractère ?
On pourrait redéfinir la multiplication `entier x std::string` (voir plus loin) : la fonction compilerait avec modif. mineures !

```
axpy(4,2,"toto"s,"titi"s); // compile si operator * défini
```

Fonctions génériques et référence universelle

- Utilisation de la double référence
- Mais pas la même signification que pour des fonctions non génériques
- Appelées en anglais soit « universal reference » soit « forwarding reference » (nom officiel)
- Agit à la fois comme une référence normale (lvalue) et comme une référence « rvalue »

```
void set_data( auto& u, auto&& temp)
{
    temp.swap(u);
}

std::vector u{1.,2.,3.,4.};
std::vector v{-1.,-2.,-3.,-4.};
set_data(u, std::vector{1.,2.,5.,6.,3.,7.});// OK, lvalue + rvalue
set_data(u, v); // OK, lvalue + lvalue !
```

Valeurs par défaut

- Arrive souvent qu'un paramètre ait quasiment toujours la même valeur
- Exemple :

```
void axpy(int N, auto a, const auto *x, auto *y, int incx, int incy)
{
    for( int i = 0; i < N; ++i ) y[i*incy] += a * x[i*incx];
}
int main()
{
    const int N = 4 ;
    double A[N][N] = { {1 ,2 , 4 ,  8},
                       {1 ,3 , 9 , 27},
                       {1 ,4 ,16 , 64},
                       {1 ,5 ,25 ,125} } ;
    // On soustrait 4 fois la colonne 1 à la colonne 3 de la matrice :
    axpy (4,-4.,A, A+2, N, N);
    // Rajout de la deuxième colonne à la quatrième ligne :
    axpy (4, 1.,A+1,&A[3][0],N,1);
```

Valeurs par défaut (suite)

- `incx` et `incy` indispensable mais égal à 1 en général
- Paramètre inutile pour les vecteur, alourdit le code…
- On leur donne une valeur par défaut égal à un
- Si on omet de les données, ils seront égaux à un !

```
void axpy(int N,auto a,const auto *x,auto *y,int incx = 1,
          int incy = 1)
{
    for (int i=0; i<N; ++i) y[i * incy] += a*x[i * incx];
}
int main()
{
    const int N = 4;
    double A[N][N] = { {1 ,2 , 4 ,  8}, {1 ,3 , 9 , 27},
                       {1 ,4 ,16 , 64}, {1 ,5 ,25 ,125}};
    double x[N] = {1 ,2 ,3 ,4}, y[N] = {4 ,3 ,2 ,1};
    axpy (N,-4.,&A[0][0],&A[0][2], N, N); // incx = N, incy = N
    axpy (N, 1.,&A[0][1],&A[3][0], N);    // incx = N, incy = 1
    axpy (N, 1.,&A[1][0],&A[0][3], 1, N); // incx = 1, incy = N
    axpy (4, 1., x, y); // incx = 1 , incy = 1;
    return EXIT_SUCCESS ;
}
```

Valeurs par défaut (suite)

- Les paramètres ayant des valeurs par défaut doivent **impérativement** être déclarée en dernier dans les paramètres de la fonction ;
- L'ordre des paramètres par défaut doit être respecté à l'appel : si un paramètre possédant une valeur par défaut doit être défini avec une valeur spécifique, **tous les paramètres précédents**, même ceux ayant une valeur par défaut, doivent également avoir une valeur spécifique définie par l'utilisateur. Ainsi, dans l'exemple ci–dessus, on ne peut pas définir une valeur différente de un pour incy sans définir explicitement la valeur un pour incx à l'appel !
- Les valeurs par défauts sont uniquement définis dans la déclaration de la fonction, pas dans la définition.

Exercice

- Écrire une petite bibliothèque d'algèbre linéaire générique
  - Écrire un produit scalaire générique avec un incrément en x et y par défaut valant un
  - Écrire une homothétie générique avec un incrément en x valant par défaut un
  - Calcul générique de la norme d'un vecteur avec incrément par défaut valant un
- Tester votre bibliothèque avec des réels simple et double précision ainsi qu'avec des complexes
- Quels soucis rencontrez-vous ? Pourquoi ?

Gestion des erreurs en C++

- En C, on rajoute un code d'erreur aux fonctions :
- Demande une stratégie d'erreur à élaborer
- En C++, on peut utiliser les exceptions (même principe qu'en Python)
- L'idée est lorsqu'on rencontre une erreur « exceptionnelle », de lancer une erreur
- Tant que cette erreur n'est pas rattrapée, elle remonte la pile d'appel
- Si elle n'est jamais rattrapée, le programme s'arrête avec une erreur
- Attention, il ne s'agit pas de rattraper des erreurs dus à des bogues, mais bien des erreurs « exceptionnelles »
- Exemple : Disque dur plein, panne réseau, etc.

Gestion des erreurs en C++

- Dans la bibliothèque **`stdexcept`** (mais pas que), plusieurs exceptions courantes sont déjà proposées

```
auto
normalise(std::vector<double> const& u)
{
  double nrm_u = normeL2(u);
  if (nrm_u < 1.E-14)
  {
    std::string msg_err =
                "Vecteur nul. norme L2 = "s +
                std::to_string(norme_u) + "\n";
    throw std::runtime_error(msg_err);
  }
  return scal(1./norme_u, u);
}
```

```
std::vector u{1.,1.,0.},v{0.,0.,0.};
std::vector u_nrm ;
try {
  u_nrm = normalise(u);
  std::cout << "u normalisé : ";
  for (const auto& val : u_nrm)
     std::cout << val << " ";
  std::cout << std::endl;
  auto v_nrm = normalise(v);
  std::cout << "v normalisé : ";
  for (const auto& val : v_nrm)
     std::cout << val << " ";
  std::cout << std::endl;
} catch(std::runtime_error& err)
{
   std::cout << err.what()
             << std::endl;
}
catch(std::logical_error& err)
{
     ...
     throw err ;
```

Gestion des erreurs en C++

- Voir `stdexcept` pour avoir les types d'exceptions prédéfinies
- On peut créer ses propres exceptions (mais il faut attendre le chapitre sur l'objet pour savoir le faire)
- Mécanisme assez lourd en temps CPU
- Mettre de gros blocs d'instructions dans les blocs d'instructions gérés par try

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

Mauvais, peu performant !

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

Bien, peu de pénalité en performance

Exercice

- Écrire une fonction d'orthonormalisation d'une famille de vecteur utilisant l'algorithme de Gram-Schmidt
- Gérer les erreurs éventuelles rencontrées à l'aide des exceptions (exemple : famille liée)