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

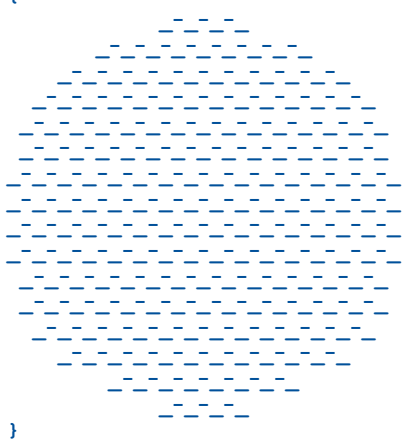
- **Vie d'un logiciel** : plus de temps à le lire qu'à programmer
- Code clair et agréable à lire : très important
- Analogie entre l'écriture d'un texte et d'un code :
  - Bien écrit
  - Bien présenté
  - Pas de fautes d'orthographe
  - Phrases bien structurées
  - Idées bien organisées et successions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

Exemple de « mauvais » code

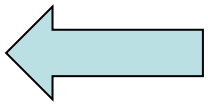
```
int k(int i)
{int rsflkj=1; if (i==1)
return rsflkj; else rsflkj = i;
return rsflkj*k(i-1);}
```

Que fait cette fonction ?

```
#define _ -F<00||--F-00--;
int F=00,00=00;main(){F_00();printf("%1.3f\n",4.*-F/00/00);}F_00()
{
```



```
}
```



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
   assert(std::abs(v[0]*u[0]+v[1]*u[1]+v[2]*u[2] )<1 .E–14) ;
}
```

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 ;
```

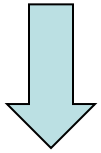
```
x = 4 ;
x += 1 ;
```

- 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, π, π^{-1}
  - 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 ’)
          << value2 << std::endl;
```

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

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

Les fonctions en C++

Surcharges des fonctions

- Plusieurs fonctions peuvent avoir le même nom du moment que les paramètres diffèrent et ne laisse 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 (
{
    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)

- Ecrire 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
```

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, allourdit 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.

Référence et passage par référence