

C++ Moderne

Introduction au C++ 11/14/17/20

Xavier JUVIGNY, SN2A, DAAA, ONERA

xavier.juvigny@onera.fr

Formation C++

- 8 Septembre 2022 -

¹ ONERA, ² DAAA

Plan du cours

- 1 Prérequis, évolution du C++ et philosophie et ambition du cours
- 2 Les cosmétiques du C++ moderne
- 3 Gestion de la mémoire depuis C++ 11
- 4 Nouveautés sur les fonctions
- 5 Nouveautés en programmation orienté objet
- 6 Conclusion première partie

Sommaire

- 1 Prérequis, évolution du C++ et philosophie et ambition du cours
- 2 Les cosmétiques du C++ moderne
- 3 Gestion de la mémoire depuis C++ 11
- 4 Nouveautés sur les fonctions
- 5 Nouveautés en programmation orienté objet
- 6 Conclusion première partie

Prérequis

Prérequis

Une bonne expérience en programmation C++ 98, en particuliers avoir une bonne notion sur :

- ➔ Les références et le bon usage du `const` ;
- ➔ L'héritage et le mécanisme des fonctions virtuelles en C++ ;
- ➔ Les templates et la généricité
- ➔ La bibliothèque standard STL du C++ (98)

Passé et avenir du C++

Passé

- 1980** : Invention du C++ par Bjarne Stroustrup : basé sur le C, s'inspire de Simula67 et Algol68
- 1998** : Première normalisation du C++ par l'ISO : Programmation orienté objet, templates et STL
- 2011** : Màj de la normalisation : Boucles basées sur des intervalles, délégation/héritage constructeurs, default et delete, déplacement, templates variadiques, fonctions lambda, etc.
- 2014** : Màj mineure de la norme : déduction automatique type retour, séparateurs de chiffre, fonctions lambdas génériques, etc.
- 2017** : Màj de la norme : if "statique", optional, déduction automatique template, enrichissement STL, etc.
- 2020** : Màj de la norme : concepts, consteval, span, ranges, fonctions génériques, initiation nommée de structures, etc.

Passé et avenir du C++...

Avenir

2023 : Màj de la norme : tableaux multiindices, covariance, extension fonctionnalités des ranges,...

2026 : Màj de la norme : ?

Les tendances

- Enrichissement de l'aspect générique du C++
- Enrichissement de l'aspect fonctionnel du C++
- Très peu d'enrichissement de l'aspect objet du C++
- Beaucoup de nouveautés pour écrire un code plus lisible et plus concis.

Philosophie et ambition de la formation

Philosophie/Ambition de ce cours

- Rechercher à créer des interfaces expressives et compréhensibles ;
- Ecrire du code simple et robuste : utiliser ce que nous proposent les normes actuelles ;
- Ne pas utiliser les nouveautés pour créer du code intellectuellement satisfaisant mais inutilement ou trop compliqué ;
- Lorsqu'il est nécessaire d'écrire du code compliqué :
 - Justifier pourquoi on ne peut pas faire du code simple ;
 - bien documenter et expliquer clairement ce qu'on fait ;
 - faire une interface simple (même si le code est compliqué).

Sommaire

- 1 Prérequis, évolution du C++ et philosophie et ambition du cours
- 2 Les cosmétiques du C++ moderne**
- 3 Gestion de la mémoire depuis C++ 11
- 4 Nouveautés sur les fonctions
- 5 Nouveautés en programmation orienté objet
- 6 Conclusion première partie

Écriture binaire et séparateur de chiffres (C++ 14)

- Écriture binaire en préfixant par `0b`;
- Possibilité de séparer les chiffres d'un nombre arbitrairement avec le symbole `'`.

Exemple de déclaration binaire et de séparateurs

```
std::int16_t bvalue = 0b0111'1010'0101'0011;
std::int16_t hvalue = 0x7A'53;
std::int16_t dvalue = 31'315;

std::cout << "0b" << std::bitset<16>(bvalue).to_string()
          << " == 0x" << std::hex << hvalue << std::dec
          << " == " << dvalue << std::endl;

double omega = 3'546.13'779'423;
std::cout << "omega = " << std::setprecision(14) << omega
          << std::endl;
```

Gestion de conditions d'erreurs statiques

Assertions “statiques” : assertions vérifiées durant la compilation et non l'exécution.

Syntaxe

```
static_assert(condition, message); // A partir de C++ 11  
static_assert(condition)           ; // A partir de C++ 17
```

Exemple d'utilisation de l'assertion statique

```
struct UneStructure {  
#if defined(COMPILER_WITH_ERROR)  
    bool un_drapeau;  
    double un_reel;  
    bool un_autre_drapeau;  
#else
```

```
    bool un_drapeau, un_autre_drapeau;  
    double un_reel;  
#endif  
};  
static_assert(sizeof(Unestructure) <= 16);
```

Aliasing de type/template

- Avant C++ 11, seul aliasing de type possible avec `typedef`
- Depuis C++ 11, préférable d'utiliser `using` qui est :
 - Plus clair et intuitif;
 - Permet également l'aliasing de templates !

Exemples d'aliasing C++11

```
template<typename K>
using CloudOfPoints=std::vector<std::array<K,3>>;
using index_t=std::uint32_t;
```

```
index_t nx=4, ny=6, nz=3, nverts=nx*ny*nz;
CloudOfPoints<double> vertices(nverts);
```

```
for (index_t iv = 0; iv < nverts; ++iv )
```

```
vertices[iv] = { 0.1 * (iv%nx),
                 0.2 * ((iv/nx)%ny),
                 0.15* (iv/nx/ny) };
```

```
for ( index_t iv = 0; iv < nverts; ++iv )
    std::cout << "( " << vertices[iv][0]
               << ", " << vertices[iv][1]
               << ", " << vertices[iv][2] << " ) ";
```

Encodage des chaînes de caractères

Depuis C++ 11, possibilité d'encoder une chaîne de caractère dans un format spécifique. On préfixe la chaîne de caractère avec l'encodage voulu :

- Codage utf8 : `u8"âprêté"` ;
- Codage "wide" : `L"âprêté"` ;
- Codage utf16 : `u"âprêté"` ;
- Codage utf32 : `U"âprêté"` ;

Possibilité de suffixer une chaîne de caractère pour définir une `std::string` au lieu d'un `char const*`.

Exemple d'utilisation du suffixe

```
#include <string>
// Indispensable pour utiliser le suffixe
using namespace std::string_literals;
std::string chaine = "Une chaîne de caractere."s;
```

Chaîne de caractère “brut”

Possibilité de définir des chaînes de caractères non interprétées.

Exemples de chaînes de caractère brut

```
const char* penguins = R"DELIM(  
    (o_  
(o_    /\n  
(/)_    v_/_ )DELIM";  
std::u8string truc = u8R"(é"o"é)";
```

Attention

- Avant C++ 20, le type de `u8"` est `char *`;
- Après C++ 20, le type de `u8"` est `char8_t*`.

enumérations (C++11/C++17)

Nouveautés

- Possibilité de définir le type contenant l'énuméré ;
- Possibilité de définir directement des énumérés avec portées ;

Exemple C++98

```
enum eColor { red, green, blue };  
eColor colour1 = green;  
eColor colour2 = red;  
eColor col{static_cast<eColor>(42)}; // Déconseillé mais possible...  
std::cout << "Colors : " << static_cast<int>(colour1) << "/"      /* 1 */  
                << static_cast<int>(colour2) << "/"      /* 0 */  
                << static_cast<int>(col) << std::endl;  /*45 */
```

Exemples énumérés en C++ 11

Exemples

```
enum class Altitude { haut, milieu, bas };
enum class Axis : char { X='X', Y='Y', Z='Z' };
struct Navigation
{
    enum direction : char { ouest = 'O', nord = 'N', est = 'E', sud = 'S' };
};

std::ostream& operator << (std::ostream& out, Axis axe ) {
    out << static_cast<char>(axe);
    return out;
}

char dir1 = Navigation::direction::nord; // A partir de C++ 11
char dir2 = Navigation::sud; // Déjà ok en C++ 98

Axis axe1 = Axis::Z; // Valeur prédéfinie d'un axe
```

Exemples énumérés en C++ 17/20

Exemple C++ 17/20

```
enum eColor { red, green, blue };
enum class Altitude { haut, milieu, bas };
enum class Axis : char { X='X', Y='Y', Z='Z' };

struct Color
{
    using enum eColor;
};

#if __cplusplus >= 202002L
    Color colors;
    eColor colour2 = colors.red;
#endif

Altitude alt{4};
Navigation::direction dir3{'H'}; // On rajoute une notion de haut...
// eColor    col{45}; //<- Erreur de compilation
eColor col{red};
// On définit un nouvel axe pour faire de la 4D :
Axis axe2{'W'}; // OK en C++17 : {} obligatoires !
```


Exemple d'aliasing d'énuméré en C++ 20

```
enum eColor { red, green, blue };  
  
struct ForegroundColor  
{  
    using enum eColor;  
};  
  
struct BackgroundColor  
{  
    using enum eColor;  
};  
  
eColor c1 = ForegroundColor::red;  
eColor c2 = BackgroundColor::blue;
```

Espaces de nommages inline

Un espace de nommage préfixé par le mot clef `inline` sont traités comme des membres de l'espace de nommage parent.

Peut être utilisé comme un mécanisme de contrôle de version :

```
namespace algebra
{
    namespace old_version
    {
        class Matrix { ... };
    }
    inline namespace new_version
    {
        class Matrix { ... };
    }
}

algebra::old_version::Matrix oldA;
algebra::Matrix newA;
```

Déclaration des structures

- À partir du C++ 11, possibilité de déclarer une structure/classe locale à une fonction.
- Permet de ne pas “polluer” l’espace de déclaration globale par une structure globale.
- La structure n’est visible que localement.

Exemple

```
std::vector<Point2D> convexHull( std::vector<Point2D> const& t_cloudOfPoints )
{
    struct Helper
    {
        enum eTurn { eLeft = 0, eRight = 1 };
        Point2D pivot, currentPoint;
        ...
        Helper( std::vector<Point2D> const& t_cloudOfPoints );

        eTurn segmentTurn( Point2D const& t_candidate ) { ... }
    };
    Helper help(t_cloudOfPoints);
    ...
}
```

Initialisation des variables (C++ 11)

Différentes syntaxes pour l'initialisation

- Possibilité d'initialiser comme en C++ 98
- Possibilité d'initialiser avec des accolades { }
- Permet dans certains cas une lecture plus facile du code !

Exemple d'initialisation avec accolades

```
struct Product { std::string nom; double prix; };  
bool addProductAt( int t_index, Product const& t_product ) { ... }  
...  
if (addProductAt( 3, {"Bouteille", 1.50 } ))  
{  
    ...  
}
```

Initialisation à la volée de la valeur de retour d'une fonction

Initialisation “à la volée” de la valeur de retour

- Retourne les paramètres d'instanciation entre accolades { }
- Permet de simplifier et de rendre le code plus lisible ;
- Dans ce cas, plus obligé de préciser le type de retour avant l'accolade.

Exemple d'utilisation

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

Initialisation par défaut de champs/attributs (C++ 11)

Principe

- Possibilité d'attribuer une valeur par défaut à un champs/attribut à une structure/classe lors des instanciations
- Pour les classes, mécanisme permettant un code plus simple et plus sûr que de devoir initialiser tous les attributs à chaque constructeur !

Exemple pour une matrice carrée “factorisable”

```
class SquareMatrix {  
public:  
    SquareMatrix() = default;  
    SquareMatrix( int dim )  
        : m_dimension(dim),  
          m_coefficients(dim*dim)  
    {}  
};
```

```
...  
private:  
    int m_dimension = 0;  
    std::vector<double> m_coefficients{};  
    std::vector<int> m_pivots{};  
};
```

Initialisation nominative de champs/attributs (C++ 20)

Principe

- Initialise que certains champs d'une structure en les nommant ;
- Il faut néanmoins respecter l'ordre de déclaration ;
- Vient du langage C 99

Exemple d'utilisation pour paramètres nommés

```
struct GmresOptions
{
    int numberRhs = 1;
    double const* preconditionner{nullptr};
    double relativeError = 1.E-6;
    std::uint32_t numberOfOuterIterations = 1;
    std::uint32_t numberOfInnerIterations = 20;
    bool isVerbose = false;
```

```
};

std::vector<double>
gmres(int dimension, double const* A, double const* b,
      GmresOptions const& options ) {
    std::vector<double> res = gmres(3, A, b,
                                   {.relativeError=1.E-14, .isVerbose=true}
                                   );
```

Déduction automatique implicite de type

Utilisation du mot clef auto

- Permet de déduire "implicitement" le type d'une variable (C++ 11) ou d'un retour (C++ 14);
- Le type de retour ne doit pas être ambigu !
- Ne pas en abuser sous peine d'un code illisible !

Exemples d'utilisation d'auto

```
template<typename K1, typename K2>
auto sub( K1 const& v1, K2 const& v2 )
{ return v1-v2; }

int main() {
    std::vector vals{ 1.5, 1.4, 1.3 };
}
```

```
for (auto it=vals.begin(); it!=vals.end(); ++it)
    std::cout << sub(int(*it),3) << " ";
    << sub(*it,2) << " ";
    << sub(2,*it) << std::endl;
```


Déduction automatique explicite de type

Utilisation du mot clef `decltype`

- Permet de déduire un type à partir d'une variable donnée ou d'une expression ;
- Utile quand la déduction implicite échoue ;
- Permet de simplifier le type de retour d'une méthode de classe ;

Comportement de `decltype`

Soient : `int i` ; `const int&& f()` ; `struct A { double x ; }` ; `const A* a = new A` ;

On a alors les règles suivantes :

- `decltype(f()) x` ; : x est une r-value (`const int&&`)
- `decltype(i) x` ; : x est une l-value (`int`)
- `decltype(A->x)x` ; : x est une l-value (`double`)
- `decltype((i)) x` ; : x est une référence (`const int&`)

Exemple d'utilisation de decltype

Exemple d'utilisation de decltype

```
template<typename K1, typename K2>
struct MaxMeanReturn_t
{
    using value_t = decltype(K1(0)+K2(0));
    value_t meanMax = std::numeric_limits<value_t>::lowest();
    std::size_t index = -1;
};

template<typename K1, typename K2> MaxMeanReturn_t<K1,K2>
searchMaxMean( std::vector<K1> const& u1, std::vector<K2> const& u2 )
{
    assert(u1.size() == u2.size());
    using value_t = MaxMeanReturn_t<K1,K2>::value_t;
    MaxMeanReturn_t<K1,K2> maxCurrent;
    for ( std::size_t index = 0; index < u1.size(); ++index )
    {
```

```
        value_t val = (u1[index]+u2[index])/value_t(2);
        if (val > maxCurrent.meanMax){
            maxCurrent.meanMax = val;
            maxCurrent.index   = index;
        }
    }
    return maxCurrent;
}

int main()
{
    std::vector u{1.,-2., 6., 4., -3. };
    std::vector v{1 , 4 ,-3 , 8 , 10 };
    auto ret = searchMaxMean(u, v);
    std::cout << "Max Mean value : " << ret.meanMax
               << " at " << ret.index << std::endl;
```

Liste d'initialisation

Definition

Liste d'initialisation : valeurs définie entre un { et un }.

Example

```
double a[] = { 1., 2., 3., 4. };
```

À partir de C++ 11

Une liste d'initialisation est un type template à part entière : `std::initializer_list`

Liste d'initialisation en C++ 11

Utilisation d'une liste d'initialisation

- Permet d'initialiser un objet, un tableau, une valeur de retour
- Utiliser en temps que type même, en paramètre d'une fonction, etc.

Exemples

```
template<typename K> K
max( std::initializer_list<K> const& t_values )
{
    auto max = *t_values.begin();
    for (auto it = t_values.begin();
         it != t_values.end(); ++it) {
        max = (max < *it ? *it : max);
    }
```

```
    }
    return max;
}

int main() {
    auto maxVal = max({1,3,-3,6,2,4});
    std::cout << "maxVal : " << maxVal << std::endl;
}
```

Boucles basées sur une plage

Définition

Plage : ensemble partiel ou non des éléments constituant un conteneur

A partir de C++11, possibilité de parcourir les éléments de tout objet possédant une méthode `begin` et une méthode `end`

L'itérateur de la boucle prend successivement toutes les valeurs itérées.

Exemple

```
for ( int prime : { 2, 3, 5, 7, 11, 13, 17} )  
{  
    std::cout << prime << "\t";  
}
```

Boucle sur une plage (suite)

Attention à l'usage de `auto` dans la boucle :

- Possibilité d'utiliser `auto` au lieu du type des valeurs contenues ;
- Par défaut, provoque une copie de la valeur courante pointée par l'itération ;
- Penser à utiliser des références ou des références constantes.

Exemple

```
std::vector<Geometry::Point3D> cloudOfPoints;
...
// On parcourt tous les points du nuage :
for (auto const& point : cloudOfPoints)
{
    ...
}
```

Boucle sur une plage avec initialisation (C++ 20)

Utile pour initialiser une valeur temporaire servant dans la boucle
Avant C++ 20, nécessité d'avoir une variable non locale à la boucle. Depuis C++ 20, possibilité que cette variable ne soit visible que par la boucle.

Exemple

```
std::vector<int> compute_some_values() {  
  
    ...  
  
    for ( int i = 0; int prime : {2,3,5,7,11,13,17,21,23,27}) {  
        std::cout << ++i << "e nombre premier : " << prime << std::endl;  
    }  
  
    for (std::vector<int> values = compute_some_values(); int value : values ){  
        std::cout << value << " ";  
    }  
}
```

Définition variable au sein de la condition d'un if (C++ 17)

Depuis C++ 17, possibilité de définir une variable dans le test conditionnel.

Avantage : La variable est locale au test.

Exemple

```
std::map<int,std::string> keys{ {2,"def"}, {3, "loop"}, {5, "cond"}, {7, "<-" } };  
if ( auto it = keys.find(11); it != keys.end() )  
    std::cout << "Longueur clef n°11 : " << it->second.size() << std::endl;  
keys[11] = "equal";  
if ( auto it = keys.find(11); it != keys.end() )  
    std::cout << "Longueur clef n°11 : " << it->second.size() << std::endl;
```


Les attributs de compilation

Moyen standardisé d'annoter du code C++.

Remplace les solutions individuelles des compilateurs : `#pragma`, `__declspec()` (Visual C++) ou `__attribute__` (g++).

Syntaxe : `[[<attribut>]]` où `<attribut>` est un attribut de la norme ou non.

Liste des attributs standards

- `[[noreturn]]` : Spécifie qu'une fonction ne doit jamais rendre la main !
- `[[deprecated("message")]]` : Wwarning pour fonction/classe/espace nommage/variable obsolète. (C++ 14)
- `[[nodiscard]]` : Valeur retournée par fonction ne doit pas être ignorée par l'appelant. (C++ 17)
- `[[may_unused]]` : Supprime avertissement pour une variable/fonction/argument de fonction non utilisé.(C++ 17)
- `[[fallthrough]]` ; : Supprime avertissement si `break` omis volontairement pour un `case`.
- `[[likely]]` : Indique qu'une condition presque toujours vraie ; (C++ 20)
- `[[unlikely]]` : Indique qu'une condition presque toujours fausse (C++ 20) ;
- `[[no_unique_address]]` : Pour structure vide, indique qu'elle ne prendra pas de place mémoire en tant qu'attribut. Sinon indique que des valeurs peuvent être stockées dans espace mémoire supplémentaire dû à l'alignement.

L'attribut [[noreturn]]

```
[[noreturn]] void f() {  
    // Bien que possédant l'attribut [[noreturn]], cette  
    // fonction retourne à l'appelant. Il y aura donc  
    // un warning généré par le compilateur :  
    // "'noreturn' function does return"  
}  
  
void g() {  
    // Fonction sans l'attribut [[noreturn]] et rendant la main.  
    // Il n'y aura pas de warning, puisqu'on lui permet de  
    // rendre la main.  
    std::cout << "Code is intended to reach here";  
}  
  
[[noreturn]] void h() {  
    // Cette fonction ne rend jamais la main et donc je retourne jamais à l'appelant.  
    // Il n'y aura donc pas de warning à la compilation.  
    while (true)  
    {}  
}
```

L'attribut [[deprecated]]

```
namespace algebra {  
    namespace deprecated {  
        struct [[deprecated]] Matrix {  
  
        ...  
  
        [[deprecated("Vieille version de GMRES. Plutot utiliser la nouvelle")] ]]  
        void GMRES( algebra::deprecated::Matrix const& A, std::vector<double> const& b,  
                    std::vector<double>& x, double& residu, double epsilon,  
                    int nombre_iterations_externes, int nombre_iterations_internes);  
  
        ...  
  
        void GMRES( algebra::deprecated::Matrix const& A, std::vector<double> const& b,  
                    std::vector<double>& x, double& residu, double epsilon,  
                    int nombre_iterations_externes, int nombre_iterations_internes)  
        {  

```

L'attribut [[nodiscard]]

```
struct Command {
public:
    Command() = default;
    ~Command() = default;

    [[nodiscard]]
    Command& begin( std::string const& promptBanner ) {
        m_prompt = promptBanner + " >";
        return *this;
    }

    [[nodiscard]]
    Command& addDescriptor() {
        std::cout << m_prompt << " Descriptor added" << std::endl;
        return *this;
    }

    [[nodiscard]]
    Command& addDrawCommand() {
        std::cout << m_prompt << " Draw command" << std::endl;
        return *this;
    }
};
```

```
    }

    void end() {
        m_prompt = "";
    }

private:
    std::string m_prompt{};
};

int main() {
    Command command;
    command.begin("New prompt")
        .addDrawCommand()
        .addDescriptor()
        .addDrawCommand()

        .end();

    command.begin("Other prompt");
    command.addDescriptor();
    command.end();
}
```

Les attributs `[[maybe_unused]]` et `[[fallthrough]]`

```
[[maybe_unused]]
void uneFonction( std::list<double> const& u )
{

...
enum MatrixProperty {
    eRectangular,
    eSquare,
    eSymmetric,
    ePositiveDefined
};
switch(t_property)
{
```

```
case ePositiveDefined:
    std::cout << "Factorisation Cholesky possible." << std::endl;
    [[fallthrough]];
case eSymmetric:
    std::cout << "Factorisation de Crout possible." << std::endl;
    [[fallthrough]];
case eSquare:
    std::cout << "Factorisation de Gauss possible" << std::endl;
    break;
default:
    std::cout << "Factorisation QR seulement possible..." << std::endl;
}
```

Attribut `[[likely]]` et `[[unlikely]]` (C++ 20)

Aide le compilateur à optimiser.

Exemple

```
double pow(double x, long long n) {  
    if (n > 0) [[likely]]  
        return x * pow(x, n - 1);  
    else [[unlikely]]  
        return 1;  
}
```

L'attribut `[[no_unique_address]]`

Pour des structures/classes sans attributs, permet de les utiliser dans une autre structure/classe sans qu'ils prennent de l'espace mémoire.

Exemple

```
struct Couleur {  
    enum eValue {  
        eBlack = 0,  
        eBlue  = 1,  
        eGreen=2,  
        eRed   = 4  
    };  
};  
class NodeAllocator {  
public:
```

```
    NodeAllocator() = default;  
    double* allocate( std::size_t t_size ) { return nullptr; }  
};  
  
struct GLNode2 {  
    [[no_unique_address]] NodeAllocator allocator;  
    double coords[3];  
    [[no_unique_address]] Couleur color;  
};
```

Sommaire

- 1 Prérequis, évolution du C++ et philosophie et ambition du cours
- 2 Les cosmétiques du C++ moderne
- 3 Gestion de la mémoire depuis C++ 11**
- 4 Nouveautés sur les fonctions
- 5 Nouveautés en programmation orienté objet
- 6 Conclusion première partie

Le pointeur nul

Nouveauté du C++ 11

- `nullptr` est le nouveau mot clef pour le pointeur nul ;
- Possède son propre type : `std::nullptr_t` ;
- Utile pour spécialisation template !

Exemple d'utilisation

```
template<typename K> int  
sizeElement( K const* ptr ) { return sizeof(K); }  
  
int sizeElement( void const* ptr) { return 1; }  
  
int sizeElement( std::nullptr_t ptr ) { return 0; }
```

Les rvalues (C++ 11)

Notion de rvalue

- Une **rvalue** est une référence à un objet temporaire (par exemple, un objet retourné par une fonction ou un objet créé au vol sans affectation à une variable).
- Une **rvalue** ne peut être affectée à une variable, sauf pour l'argument d'une fonction.
- La syntaxe d'une **rvalue** est `type&& nom` (sauf si la fonction est une fonction template...)

Exemple

```
void fonction1( std::vector<double>&& vecteur_temporaire )  
{ // On peut détruire les données du vecteur dans la fonction,  
  // vu qu'il est temporaire !  
}
```

Les opérations de déplacement

Notion

- En C++ 98, on ne pouvait que copier un objet temporaire (rvalue) ;
- À partir de C++11, possibilité de "voler" les données de l'objet temporaire.
- Le déplacement vole les données de l'objet temporaire qui se retrouve dans un état indéterminé.
- **Exemple** : Un déplacement pour `std::vector` consiste à échanger les deux pointeurs sous-jacents.
- `std::move` permet de provoquer le déplacement des données d'une variable (non constante) ;
- Un déplacement à lieu lorsqu'on retourne une variable locale d'une fonction, qu'on affecte un objet temporaire à une variable, etc.
- On verra plus tard comment définir des opérations de déplacement pour nos propres classes.

Exemples d'opérations de déplacement

Copie contre déplacement

```
std::vector<double> f() {  
    std::vector ret{3.,2.,1.};  
    return ret;  
}  
  
std::vector<double> const& g( std::vector<double> const& u ) { return u; }  
  
std::vector<double> a{1.,2.,3.};  
std::vector<double> b = std::vector{2.,3.,4.}; // Opération de déplacement  
std::vector<double> c = a; // Opération de copie;  
std::vector<double> d = std::move(c); // Opération de déplacement, c n'est plus "utilisable"  
std::vector<double> x = f(); // Déplacement dans e (retour vecteur local)  
std::vector<double> y = g(); // Copie dans y (retour vecteur global)
```

Retour par valeur d'une fonction

- 👉 La norme C++ impose qu'une variable locale à une fonction retournée par valeur par la fonction sera déplacée ;
- 👉 Aucune pénalité mémoire ou CPU à retourner un `std::vector`, une liste, un dictionnaire ou tout objet dont l'opération de déplacement n'est pas coûteux.
- ✍ **Attention**, le coût de déplacement d'un tableau statique est linéaire en fonction de la taille du tableau : le déplacement dans ce cas est en fait une copie !
- ✍ Si l'objet retourné ne possède pas d'opérateur de déplacement, une copie est effectuée en lieu et place de l'opération de déplacement.
- Bien se documenter sur le type relatif à l'objet renvoyé !

Un exemple concret d'utilisation des rvalues

```
class Vecteur
{
...
    Vecteur operator + ( Vecteur const& u ) const {
        std::cout << __PRETTY_FUNCTION__ << std::endl;
        assert(this->dimension() == u.dimension());
        Vecteur res(u.dimension());
        for (std::size_t i=0; i<u.dimension();++i)
            res[i] = m_coefs[i] + u[i];
        return res;
    }

    Vecteur&& operator + ( Vecteur&& u ) const {
        std::cout << __PRETTY_FUNCTION__ << std::endl;
        for (std::size_t i=0; i<u.dimension();++i)
            u[i] += m_coefs[i];
        return std::move(u);
    }
...
};
```

```
};

Vecteur&& operator + ( Vecteur&& u, Vecteur const& v ) {
    std::cout << __PRETTY_FUNCTION__ << std::endl;
    for (std::size_t i=0; i<u.dimension();++i)
        u[i] += v[i];
    return std::move(u);
}

Vecteur&& operator + ( Vecteur&& u, Vecteur && v ) {
    std::cout << __PRETTY_FUNCTION__ << std::endl;
    for (std::size_t i=0; i<u.dimension();++i)
        u[i] += v[i];
    return std::move(u);
}

int main() {
    Vecteur u({1.,2.,3.,4.,5.}), v({2.,4.,6.,8.,10.});
    Vecteur w({-1.,-2.,-1.,-2.,-1.}), x({ 0.,1.,0.,1.,0.});
    Vecteur y({ 1.,0., 1., 0., 1.});
    Vecteur z = u + v + w + x + y;
```

Les pointeurs partagés

Caractéristiques

Pointeur intelligent conçu pour partager avec d'autres pointeurs un même objet et responsable de la destruction de cet objet.

Cet objet sera détruit quand :

- le dernier pointeur partagé pointant sur l'objet est détruit ;
- le dernier pointeur partagé pointant sur l'objet est affecté à un autre objet ou au pointeur nul.

Exemple d'utilisation

```
#include <memory>
class GraphicDevice;
class Window
{
public:
    Window() :
```

```
        m_device(std::make_shared<GraphicDevice>())
    {}
private:
    std::shared_ptr<GraphicDevice> m_device;
};
```

Les pointeurs partagés

Remarques

- La destruction d'un objet pointé par un pointeur partagé est géré par le C++ ;
- `std::make_shared` prend en argument les arguments nécessaires à la construction de l'objet.
- s'utilise comme un pointeur classique.
- Compatible avec le mécanisme des méthodes virtuelles

Exemple

```
class Shape { ...  
    virtual double computeVolume() = 0;  
};  
class Sphere : public Shape { ... };  
class Tetrahedre : public Shape { ... };  
...
```

```
std::vector<std::shared_ptr<Shape>> shapes;  
shapes.push_back(  
    std::make_shared<Sphere>( {1., 2., 3.}, 4. );  
...  
auto volume = shapes[0]->computeVolume();
```


Pointeurs partagés

Pointeur "faible" : `std::weak_ptr`

- 👉 Pointeur intelligent faisant une référence faible sur un objet géré par un pointeur partagé.
- 👉 Référence faible : le pointeur ne gère pas le cycle de vie de l'objet pointé.
- ✍ Si le pointeur partagé est détruit avant le pointeur faible, ce dernier crée un pointeur partagé temporaire jusqu'à sa propre destruction ;
- ✍ Permet en particuliers d'empêcher des cycles d'objets se faisant références via les pointeurs partagés : on transforme un des pointeurs du cycle en pointeur faible.

Pointeur unique

Pointeur garantissant à être le seul à pointer sur un objet dédié

Conséquences et propriétés

- ➔ Pointeur non copiable
- ➔ Mais uniquement déplaçable !
- ➔ Détruit l'objet pointé lors de la destruction du pointeur ou si le pointeur est affecté à une autre variable (ou à nullptr).

Exemple pratique

```
void uneFonction() {  
    std::unique_ptr<Mesh> a_mesh(new CartesianMesh( {10,15,10}, {0.1,0.05,0.2} ) );  
    ...  
    auto nbVertices = a_mesh-> numberOfVertices();  
    ...  
} // a_mesh est automatiquement détruit à la sortie de la fonction.
```

Exemples d'utilisation des pointeurs intelligents

Utilisation des pointeurs uniques

```
template<typename ValueType>
struct NAryTreeNode {
    ValueType value;
    std::vector<std::unique_ptr<NAryTreeNode>> children;
};

using NAryTree = NAryTreeNode;
```

Utilisation des pointeurs partagés

```
template<typename ValueType>
struct GraphNode {
    ValueType value;
    std::vector<std::shared_ptr<GraphNode>> connectedNodes;
};

using Graph = GraphNode;
```

Conclusion sur la gestion mémoire

- ❑ L'usage des rvalues permet d'éviter des copies et des réservations mémoires inutiles ;
- ❑ Permet aux fonctions de mieux séparer arguments des résultats de sortie ;
- ❑ Pointeurs intelligents réduit grandement la complexité de la gestion de la mémoire ;
- ❑ Opérateurs `new` et `delete` rarement utilisés dans un code C++ 11...

Exemple : produit tensoriel de deux vecteurs

En C++ 98

```
void
tensorProd(Vecteur const& u, Vecteur const& v,
            PlainMatrix& tensorMatrix ) {
    // tensorMatrix déjà initialisé à l'extérieur
    ... }
PlainMatrix*
tensorProd(Vecteur const& u, Vecteur const& v) {
    PlainMatrix* tensorMat =
        new PlainMatrix(u.dim(), v.dim()); ...
    return tensorMat;
} // À l'utilisateur de faire : delete tensorMat !
```

En C++ 11

```
PlainMatrix
tensorProd(Vecteur const& u, Vecteur const& v) {
    PlainMatrix tensorMat(u.dim(), v.dim());
    ...
    return tensorMat;
}
PlainMatrix
operator * (Vecteur const& u, Vecteur const& v) {
    return tensorProd(u, v);
}
```

Sommaire

- 1 Prérequis, évolution du C++ et philosophie et ambition du cours
- 2 Les cosmétiques du C++ moderne
- 3 Gestion de la mémoire depuis C++ 11
- 4 Nouveautés sur les fonctions**
- 5 Nouveautés en programmation orienté objet
- 6 Conclusion première partie

Retour d'une valeur par liste d'initialisation

Si le type de retour est connu, il est possible de retourner la valeur par liste d'initialisation sans préciser son type.

Exemple

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

Remarque : Si il n'est pas nécessaire de préciser le type de retour en utilisant une liste d'initiation, il est par contre obligatoire le type du premier élément constituant un conteneur (array, vector, list,etc.)

Fonctions anonymes

Définition

Fonction sans nom, manipulée comme une valeur (comme un entier, réel, etc.). Appelée aussi fonction lambda en référence au lambda-calcul qui a introduit les fonctions anonymes.

Syntaxe d'une fonction anonyme en C++




Syntaxe :

`[<capture>] (<paramètres>) { Corps de la fonction anonyme } -> <type retour> où`

- `<capture>` : liste de variable visible par la fonction anonyme ;
- `<paramètres>` : paramètres à passer à la fonction anonyme (\equiv fonction classique) ;
- La déclaration du type retour est optionnel.

Fonctions anonymes (suite)

Caractéristiques d'une fonction anonyme

-  Peut être déclarée au vol dans les paramètres d'une fonction ;
-  Aucune pénalité d'appel par rapport à une fonction classique ;
-  Chaque fonction anonyme possède son propre type \Rightarrow templates obligatoire pour paramètre.

Exemples d'utilisation des fonctions anonymes

Exemple

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    auto print = [] ( std::vector<double> const& a ) { for (auto v : a ) std::cout << v << " ";
                                                         std::cout << std::endl; };

    std::vector<double> a_array{1., -3., 2., -7., 6., 4.};
    std::cout << "Tableau avant tri : "; print(a_array);
    std::sort(a_array.begin(), a_array.end(),
              [](double x, double y) { return std::abs(x) < std::abs(y); } );
    std::cout << "Tableau apres tri par valeur absolue : "; print(a_array);
    return EXIT_SUCCESS;
}
```

Remarque : Utiliser auto pour déclarer la variable print est ici obligatoire puisqu'on ne connaît pas le type de la fonction lambda affectée à la variable !

Le wrapper de fonction

Wrapper polymorphe permettant d'encapsuler une fonction de type :

- Fonction C/C++ ;
- Une fonction anonyme ;
- Toute structure possédant un opérateur d'évaluation.

Bibliothèque : `<functional>`

Syntaxe : `std::function<retour_type(parametres_types)> fvar;`

Intérêts : Permet de manipuler des fonctions anonymes, des fonctions C/C++ ou des foncteurs aisément

Désavantage : Légère pénalité lors de l'appel de la fonction via ce wrapper.

Capture d'une fonction anonyme

Définition

Capture \equiv variables externes visibles par copie/référence dans le corps de la fonction anonyme.

Plusieurs variables peuvent être capturer, séparés par une virgule entre [].

Plusieurs modes de capture possible :

- [var1] : Capture par copie une variable var1 ;
- [&var1] : Capture par référence d'une variable var1 ;
- [=] : Capture par copie toutes les variables visibles ;
- [&] : Capture par référence toutes les variables visibles ;
- [&, var1] : Capture par référence toutes les variables sauf var1 capturé par copie.
- [this] : Capture *this par référence ;
- [*this] : Capture *this par valeurs.

Exemples de fonctions avec des captures

Exemples

```
class Polynome {
    auto primitive( double t_constante ) const {
        return [*this,t_constante] ( double x ) {
            double value = t_constante;
            for ( std::uint32_t iDeg = 0;
                  iDeg <= degree(); ++iDeg )
                value += std::pow(x,iDeg+1)/(iDeg+1.);
            return value;
        };
    }
}; // Fin class Polynome

auto d( Polynome const& P ) {
    return [P] ( double x )
    {
        double value = 0.;
        for (std::uint32_t i=1; i<=P.degree(); ++i) {
```

```
            value += i*P[i]*std::pow(x, i-1);
        }
        return value;
    };

    std::vector<std::uint32_t> fibCoefs;

    std::function<std::uint32_t(std::uint32_t)>
        fibonacci =
        [&fibCoefs,&fibonacci] (std::uint32_t n)
        ->std::uint32_t {
            if (fibCoefs.size() < n+1)
                fibCoefs.resize(n+1);
            if (n>0) fibonacci(n-1);
            if (n < 2) fibCoefs[n] = 1; else
                fibCoefs[n]=fibCoefs[n-1]+fibCoefs[n-2];
            return fibCoefs[n];
        };
};
```

Fonctions anonymes génériques

Deux solutions pour obtenir des fonctions anonymes génériques :

- Utiliser les templates (C++ 20)
- Utiliser le mot clef auto (C++ 14)

Exemple avec template

```
auto gen_add = [] <typename T> ( T const& a, T const& b ) { return a+b; };
```

Exemples avec auto

```
auto gen_add2 = [] ( auto const& a, auto const& b ) { return a+b; };  
  
auto fibonacci = [] (auto self, std::uint32_t n, std::uint32_t a=1, std::uint32_t b=1 ) ->std::uint32_t  
{  
    return n>0 ? self(self,n-1,a+b,a) : b;  
};  
  
for ( std::uint32_t i=0; i<20; ++i)  
    std::cout << "F_" << i << " = " << fibonacci(fibonacci, i) << std::endl;
```

Définition et déclaration de fonctions en notation fonctionnelle

Syntaxe

```
auto nom_fonction(paramètres...) -> [type_de_retour]
{
    ...
}
```

Utilité de cette syntaxe

- Peut simplifier la déclaration ou la définition d'une méthode ;
- Utile en association avec les déductions automatiques explicites ;

Exemple basique

```
template<typename K>
auto square( K const& x ) -> K
```

Notation fonctionnelle

Exemple de simplification

```
namespace Algebra {  
class Vecteur {  
public:  
    using iterator = std::vector<double>::iterator;  
  
    iterator begin();  
    iterator end ();  
private:  
    std::vector<double> m_coefficients;  
};  
}  
  
Algebra::Vecteur::iterator Algebra::Vecteur::begin()  
{ return m_coefficients.begin(); }  
  
auto Algebra::Vecteur::end() -> iterator
```

Déduction automatique de type

Intérêts de la déduction automatique de type

- Éviter des redondances d'information pour un code plus clair et plus concis ;
- Permet de simplifier la déclaration du type de la valeurs de retour d'une fonction template

Exemples "redondances" ou "compliqués" en C++ 98

```
std::vector<double> u(100);  
...  
std::vector<double>::const_iterator it = u.begin();  
//  
template<typename K1, typename K2> [type de retour ?]  
add( K1 const& val1, K2 const& val2)  
{  
    return val1 + val2;  
}
```


Utilisation de la notation fonctionnelle avec `decltype`

La notation fonctionnelle peut être indispensable pour utiliser `decltype`.

Exemple où la notation fonctionnelle est indispensable

```
template<typename K1, typename K2>
auto
absadd( K1 val1, K2 val2 ) -> decltype(std::abs(val1)+std::abs(val2))
{
    return std::abs(val1)+std::abs(val2);
}
```

Remarque 1 : Il n'est pas possible de combiner détection automatique du type de retour avec le retour par liste d'initialisation (Sinon le compilateur est incapable de deviner le type de retour).

Remarque 2 : `decltype` n'évalue pas l'expression entre ses parenthèses, mais ne fait que déduire le type de retour d'une telle expression.

Déduction implicite du type de retour (C++ 14)

À partir du C++ 14, possibilité de laisser le compilateur deviner le type de retour.

Exemple

```
template<typename K1, typename K2> auto  
absadd( K1 val1, K2 val2 ){  
    return std::abs(val1)+std::abs(val2);  
}
```

Attention : Les différents return de la fonction doivent retourner exactement le même type de valeur.

Exemple ne compilant pas

```
template<typename K> auto vabs( std::size_t nbelts, K* values ) {  
    if (nbelts == 0) return nullptr;  
    for ( std::size_t i=0; i<nbelts; ++i) values[i]=std::abs(values[i]);  
    return values;  
}
```

Déduction automatique implicite et fonctions anonymes

Le type d'une fonction anonyme n'est pas connu avant compilation \Rightarrow une fonction anonyme ne peut être passée en argument que par template.

Exemple

```
template<typename Func>
void transform( std::vector<double>& array, Func const& f )
{
    for (auto& val : array ) val = f(val);
}

std::vector u{1.,2.,3.,5.,7.,11.,13.,17.};
transform(u, [] (double x) { return x*x+1; } );
```

Déduction automatique implicite et fonctions anonymes

Chaque fonction anonyme a son propre type connu à la compilation \Rightarrow Déduction automatique implicite obligatoire dans ce cas

Conséquence : Impossible de créer une fonction retournant une fonction anonyme avant C++ 14!

Exemple

```
template<typename F1, typename F2>
auto funcDiv( F1 const& f1, F2 const& f2 )
{
    return [&f1, &f2] (double x) { return f1(x) / f2(x); };
}

auto f = funcDiv( (double (*)(double))(std::cos), [] (double x) { return std::sin(x*x+1); } );
std::cout << "F(pi) = " << f(pi) << std::endl;
```

Retour de valeurs multiples par une fonction

Pourquoi retourner plusieurs valeurs avec une fonction ?

- Cela évite d'appeler plusieurs fonctions
- Certaines valeurs peuvent se calculer plus simplement simultanément (sin et cos, indice et max, etc.)

Quelles techniques disponibles avant C++ 11 ?

- Passer les valeurs de sortie par référence : interface peu claire entre entrée/sortie et sortie pure.
- Créer une structure contenant les diverses valeurs à retourner : Attention au coût de la copie !
- Retourner un tableau si données de retour de même type. Mais attention au coût de la copie !

Retour de valeurs multiples à partir de C++ 11

Quelles techniques employer ?

- Retourner une structure
- Retourner un tableau (mais attention copie si tableau statique !) si valeurs de retour de même type ;
- Retourner une paire (si deux valeurs de retour), un tuple sinon (voir planche suivante) ;

Les tuples

Définition

Structure contenant n valeurs pouvant être de types hétérogènes (n fixé).

Le tuple en C++

- Les tuples sont définis dans la bibliothèque `tuple`.
- Les tuples définis à l'aide de template : seuls les tuples possédant les mêmes types de valeurs ordonnés sont de même type
- Syntaxe : `std::tuple<T1,T2,...,Tn> a_tuple(var1, var2,...,varn);`
- C++ propose une fonction pour créer plus facilement un tuple :
`auto a_tuple = std::make_tuple(var1, var2,...,varn);`
- Pour accéder à la i ème valeurs : `auto v2 = std::get<i>(a_tuple);`. **Remarque** : l'indice passé à `get` doit être une valeur constante.

Affectation destructurante (C++ 17)

Principe

Déstructurer un agrégat de valeurs en plusieurs variables

Exemples

```
struct Point {  
    double x,y;  
    Point(double t_x,double t_y) : x(t_x), y(t_y) {}  
};  
  
Point p(2.5,1.4);  
auto [a,b] = p;  
auto &[c,d] = p;  
auto [nom, age, taille] = std::make_tuple("Tintin"s, 32, 176);  
auto {x,y,z} = std::array{1., 3., 6.};
```

Voir [cette page canadienne](#) détaillant les affectations destructurantes

Affectations déstructurantes et retour multiple des fonctions

```
std::pair<std::size_t, double> locAbsMax( std::vector<double> const& array ) {  
  
    ...  
    return {loc, valMax};  
}  
  
auto [iloc, val] = locAbsMax(data);  
  
...  
template<int deg> std::array<std::complex<double>,deg> nthUnityRoots() {  
    static_assert(deg>0);  
    constexpr double pi=3.141592653589793;  
    double angle=2.*pi/deg;  
    std::array<std::complex<double>,deg> roots;  
    for ( int i=0; i<deg; ++i) roots[i] = std::complex{ std::cos(i*angle), std::sin(i*angle)};  
    return roots;  
}  
  
auto [r1,r2,r3] = nthUnityRoots<3>();
```

Retour par structure

Principe

- Retourner une structure contenant les diverses valeurs retournées par la fonction ;
- **Avantage** : Chaque champs de la structure donne signification à la valeur retournée ;
- **Désavantage** : Légère "Pollution" du code. Mais tuple fait de même.

Exemple

```
struct SegmentIntersectionData {  
    std::optional<Point> intersectionNode;  
    std::optional<double> t1, t2, t1min, t1max, t2min, t2max;  
    bool has_intersection{false};  
};  
  
SegmentIntersectionData  
computeSegmentIntersection( Segment const& t_1stSegment, Segment const& t_2ndSegment ) {  
    ...}  
  
    auto result = computeSegmentIntersection( s1, s2 );  
    if (result.has_intersection)
```

Retour sur l'affectation destructurante (C++ 17)

Contrainte permettant l'affectation destructurante

L'affectation structurante n'est possible que pour des classes/structures dont les attributs sont **tous** publics.

Conséquence

Retour par structure n'empêche pas utilisation affectation destructurante

Retour sur l'exemple précédent

```
auto result = computeSegmentIntersection( s1, s2 );  
if (result.intersectionNode.has_value()){ Point const& intersectPoint=result.intersectionNode.value();  
    ... }  
auto const& [intersectPt,t1,t2,t1Min,t1Max,t2Min,t2Max,hasIntersect]=computeSegmentIntersection(s1,s2);  
if (intersectPt.has_value()) { ... }
```

Fonction sans exception : `noexcept`

Principe

Déclare qu'une fonction (éventuellement anonyme) ne jette pas d'exceptions.

À quoi ça sert ?

- S'assurer que si je n'attrape pas d'exceptions, le programme ne se termine pas ;
- Nécessité que cette fonction ne jette pas d'exceptions ;
- Permet certaines optimisations lors de la compilation du code.

Intérêt de `noexcept`

- Les deux premiers points (s'assurer qu'une fonction ne jette pas d'exception) est purement informative (en fait, cela n'empêche pas de pouvoir faire un `throw...`) ;
- Seul le troisième point (optimisation du code) a un effet sur l'exécutable.

Quand utiliser `noexcept` ?

- La norme pour la STL garantit qu'après une exception (ou une erreur), l'état d'un objet retrouve l'état antérieur à l'appel ayant provoqué l'exception ;
- Après un `push_back` d'un conteneur provoquant une re-allocation, il faut copier ou déplacer les éléments de l'ancienne location dans la nouvelle ;
- Si déplacement, impossibilité de restituer l'ancien état du conteneur en cas d'exception durant déplacement ;
- `push_back` préférera copier que déplacer si déplacement peut générer une erreur...
- Rajouter `noexcept` au déplacement permet d'optimiser `push_back` !

expressions constantes et fonctions

Principe

- variable n'existant et utilisé que le temps de la compilation
- Fonction pouvant être utilisée et évaluée pendant la compilation (Si la fonction n'est utilisée que pendant la compilation, elle n'est pas référencée dans l'exécutable).

Règles des expressions constantes

- Remplace une variable `const` garantissant qu'aucune place mémoire prise à l'exécution.
- Son expression ne doit contenir que des expressions constantes ;
- Pour une fonction, évaluation à la compilation ssi évaluée pour expression constante.
- Dans le cas contraire, est évaluée comme une fonction ordinaire durant l'exécution.
- Plusieurs fonctions de la STL sont des expressions constantes. Exemple :

```
constexpr double maxValue = std::numeric_limits<double>::max();
```

Exemples d'expressions constantes et utilisations

Exemples

```
constexpr double squareRoot( double x )
{
    // Utilise la méthode de Wilkes -- Wheeler et Gill (1950)
    double alpha=1, s = x;
    while (s >= 3)
    {
        alpha *= 2; s = s/4;    }
    double a = s, c = s-1;

    while (std::abs(c)>1.E-16)
        { a = a - a*c/2.; c = c*c*(c-3)/4.; }
    return alpha*a;
}
// sqrt2 évaluée à la compilation
constexpr double sqrt2 = squareRoot(2.);
// squareRoot(3.) évalué à l'exécution :
std::cout << "sqrt(3) = " << squareRoot(3.) << std::endl;
```

Remarque : Les fonctions mathématiques ne sont pas constexpr.

Complément constexpr et consteval (C++ 20)

Problème des constexpr

- Facile d'oublié de déclarer une variable `constexpr` ;
- L'appel de la fonction `constexpr` se fait alors à l'exécution et fait parti de la table des symboles ;
- On aimerait dans certains cas que la fonction ne serve qu'à évaluer des `constexpr`.

consteval (C++20)

- On peut déclarer une fonction `consteval` au lieu de `constexpr` ;
- La fonction ne peut alors être appelée qu'à la compilation !
- Elle peut servir dans l'évaluation d'une variable ordinaire ;
- Mais la valeur de ses paramètres doivent être connues à la compilation.

Exemple de fonction consteval

```
constexpr double squareRoot( double x )
{
    double alpha=1, s = x;
    ...
    return alpha*a;
}
// sqrt2 évaluée à la compilation
constexpr double sqrt2 = squareRoot(2.);
// squareRoot(3.) évalué à la compilation :
std::cout << "sqrt(3) = " << squareRoot(3.) << std::endl;
double x = 5;
// Erreur de compilation : x n'est pas constant et donc peut avoir changé
// en une valeur non connue à la compilation (en multithreading par exemple).
double sqrtx = squareRoot(x);
```

if constexpr (C++ 23)

Problématique

- Certaines fonctions peuvent être optimisées quand évaluées à l'exécution ;
- Comment distinguer les cas où on évalue à la compilation des cas où on évalue à l'exécution ?

if constexpr

- En c++ 23, `if constexpr` permet de compiler une partie du code quand la fonction est évaluée à la compilation et une autre partie du code quand la fonction est évaluée à l'exécution.

Exemple de if consteval

```
constexpr double squareRoot( double x )
{
    if consteval
    { // Utilise la méthode de Wilkes -- Wheeler et Gill (1950)
        double alpha=1, s = x;
        ...
        return alpha*a;
    }
    else
    {
        return std::sqrt(x);
    }
}

// sqrt2 évaluée à la compilation utilisant méthode de Wilkes-Wheeler et Gill
constexpr double sqrt2 = squareRoot(2.);
// squareRoot(3.) évalué à l'exécution utilisant le hardware:
std::cout << "sqrt(3) = " << squareRoot(3.) << std::endl;
double x = 5;
// squareRoot(x) évalué à l'exécution utilisant le hardware.
double sqrtx = squareRoot(x);
```

Sommaire

- 1 Prérequis, évolution du C++ et philosophie et ambition du cours
- 2 Les cosmétiques du C++ moderne
- 3 Gestion de la mémoire depuis C++ 11
- 4 Nouveautés sur les fonctions
- 5 Nouveautés en programmation orienté objet**
- 6 Conclusion première partie

Initialisation des attributs de la classe

Principe

Donner à chaque attributs déclarés dans la classe une valeur par défaut.

Conséquence

- Allège constructeurs : initialiser seulement les attributs modifiés par le constructeur.
- Permet de modifier une classe en rajoutant un nouvel attribut de façon plus sécurisée.

Exemple

```
class SquareMatrix {  
public:  
    SquareMatrix() = default;  
    SquareMatrix( std::size_t t_dim ) : m_coefs(t_dim*t_dim) {}  
    ...  
private:  
    std::vector<double> m_coefs{};  
    bool is_factorized{false};  
};
```

Délégation constructeur

Appel d'un constructeur par un autre constructeur.

Principe

Possibilité d'appeler dans la liste d'initialisation un autre constructeur de la classe.

Exemple

```
class SquareMatrix {  
public:  
    SquareMatrix( std::size_t t_dim ) : m_coefs(t_dim*t_dim) {}  
    SquareMatrix( std::size_t t_dim, double t_fillValue )  
        : SquareMatrix(t_dim)  
    {  
        std::fill(m_coefs.begin(),m_coefs.end(), t_fillValue);  
    }  
    ...  
};
```

Constructeurs et opérateurs implicites

Les constructeurs et opérateurs implicites(C++11)

- Le constructeur par défaut : `A()` ;
- Le constructeur de copie : `A(A const&)` ;
- **Le constructeur de déplacement** : `A(A &&)` ;
- Le destructeur : `~A()` ;
- L'opérateur de copie : `A& operator = (A const&) ;` ;
- **L'opérateur de déplacement** : `A& operator = (A &&) ;` .

Les règles implicites

- Le constructeur par défaut est implicite si aucun autre constructeur n'est déclaré ;
- Constructeur & opérateur de copie implicites si constructeur & opérateur de déplacement non déclarés ;
- Constructeur & opérateur de déplacement implicites si constructeur & opérateur de copie non déclarés.

Constructeurs et opérateurs implicites par défaut ou supprimés

Principe

Possibilité de déclarer par défaut (= default) ou supprimer (= delete) un constructeur ou un opérateur implicite (**Remarque** : le destructeur ne peut pas être supprimé).

Exemple

```
class SquareMatrix {
public:
    SquareMatrix() = default;
    SquareMatrix( SquareMatrix const&) = delete;
    SquareMatrix( SquareMatrix      &&) = default;
    ~SquareMatrix() = default;
    ...
    SquareMatrix& operator = ( SquareMatrix const& ) = delete;
    SquareMatrix& operator = ( SquareMatrix      && ) = default;
... };
```

Remarque : Déplacement par défaut noexcept si tous les attributs sont déplaçables avec opérateurs noexcept.

Problématiques

- Aucune protection si on se trompe dans la signature de la fonction virtuelle surchargée ;
- Comment empêcher une fonction virtuelle d'être surchargée dans des classes filles ?
- Comment empêcher une classe fille de ne pas être elle-même héritée par une autre classe ?

Vérification surcharge une fonction virtuelle : `override`

```
class Matrix { ...  
    virtual Vector operator * ( Vector const& ) = 0;  
};  
class SparseMatrix { ...  
    virtual Vector operator * ( Vector const & u ) override;  
};
```

Fonction virtuelle finale

Principe

Permet dans une classe fille qu'une fonction virtuelle ne puisse pas être surchargée dans une classe héritant de cette classe fille.

Exemple

```
class CSRMatrix : public SparseMatrix
{...
    virtual Vector operator * ( Vector const& u ) final;
};
class CSRBlockMatrix : public CSRMatrix
{...
    // Erreur de compilation : l'opérateur est final dans la classe file !
    virtual Vector operator * ( Vector const& u ) override;
};
```

Classe finale

Principe

Classe fille déclarée finale : empêche cette classe d'être héritée.

Exemple

```
class PlainMatrix final : public Matrix {...
    virtual Vector operator * ( Vector const& u ) override;
... };
// Erreur de compilation : PlainMatrix est final !
class PlainExtensibleMatrix : public PlainMatrix
{ ...
};
```

Opérateur de conversion explicite

Principe

L'appel à l'opérateur de conversion ne peut se faire qu'explicitement par l'utilisateur

Exemple

```
class Vector {...  
explicit std::string() const { ... };  
...};  
  
Vector u; ...  
std::cout << std::string(u) << std::endl;  
// Ligne ci-dessous provoque erreur compilateur :  
std::string su = u;
```

Relation d'ordre (C++ 20)

Principe

Permet de préciser un type de relation d'ordre :

- **Relation d'ordre fort** : `std::strong_ordering`
 - Les valeurs équivalentes sont indistinguables ;
 - Toutes les valeurs sont comparables.
- **Relation d'ordre faible** : `std::weak_ordering`
 - Les valeurs équivalentes sont distinguables ;
 - Toutes les valeurs sont comparables.
- **Relation d'ordre partiel** : `std::partial_ordering`
 - Les valeurs équivalentes sont distinguables ;
 - Certaines valeurs ne sont pas comparables entre elles.

Opérateur vaisseau (C++ 20)

Principe

Définir un seul opérateur de comparaison permettant de définir la relation d'ordonnancement et quel type de relation d'ordre est utilisé : **Symbole** : `<=>`

Propriétés

- Si les opérateurs `<`, `<=`, `>`, `>=` ne sont pas définis, le compilateur utilise l'opérateur `<=>`
- Par contre, on doit définir l'opérateur `==` pour les opérateurs `==` et `!=`
- On peut également utiliser les valeurs des relations d'ordre :

```
auto cmp = a<=>b;  
if (cmp==std::partial_ordering::unordered) std::cout<<"a et b ne sont pas comparables"<<std::endl;
```

Opérateurs de comparaisons par défaut (C++ 20)

Principe

Possibilité de définir l'opérateur `<=>` par défaut

Propriétés

- Effectue une comparaison lexicographique des attributs de la classe (dans l'ordre de déclaration) et si tableau, effectue une comparaison lexicographique du tableau ;
- L'opérateur `==` est aussi défini par défaut implicitement (et optimisé pour ne pas tout comparer).

Exemple

```
struct Vecteur { ...  
    auto operator <=> ( Vecteur const& ) const = default;  
    ...  
    std::array<double,3> m_values;  
};
```

Opérateurs littéraux

Définition

- Rajout d'un symbole (suffixe) après une valeur pour définir son type ;
- Existe déjà en C : `3u`, `4.f`, `5ULL` ;
- Depuis C++ 11, possibilité de définir ses propres suffixes devant commencer par un `_`.
- **Syntaxe** : `<type retour> operator "" _<suffixe>(<liste paramètres>)`

Liste paramètres autorisés

- `(const char*)`, `(unsigned long long int)` OU `(long double)`
- `(char)`, `(wchar_t)`, `(char8_t)` (C++20), `(char16_t)` OU `(char32_t)`
- `(const K*, std::size_t)` avec `K = (char)`, `(wchar_t)`, `(char8_t)` (C++20), `(char16_t)` OU `(char32_t)`

Exemple opérateurs littéraux

```
class mètre {...  
};  
class kilomètre {...  
};  
constexpr kilomètre operator"" _km( long double dst ) {  
return kilomètre{static_cast<double>(dst)};  
}  
constexpr mètre operator"" _m( long double dst ) {  
return distance::mètre{static_cast<double>(dst)};  
}  
int main() { ...  
auto d1 = 1.2_km;  
auto d2 = 100.0_m;  
...}
```

Opérateurs littéraux prédéfinis

- Pour les `std::string` : `"Hello"s`
- Pour les `std::complex` : `3.+4.i`; `3.if+4.if`
- Pour la durée `std::chrono::duration` : `3h+4min+30s+8ms+12us+13ns` ;

Multi-indices (C++ 23)

Principe

Possibilité de définir un opérateur `[]` ayant plusieurs paramètres en arguments.

Exemple

```
class PlainMatrix {  
    using index_t = std::uint32_t;  
    ...  
    double& operator [] (index_t i, index_t j ) { return m_coefs[i+j*m_nbRows]; }  
    ...  
private:  
    index_t m_nbRows, m_nbCols;  
    double m_coefs;  
};  
...  
constexpr PlainMatrix::index_t dim = 10;  
PlainMatrix A(dim, dim);  
A[3,5] = -1.;
```

Sommaire

- 1 Prérequis, évolution du C++ et philosophie et ambition du cours
- 2 Les cosmétiques du C++ moderne
- 3 Gestion de la mémoire depuis C++ 11
- 4 Nouveautés sur les fonctions
- 5 Nouveautés en programmation orienté objet
- 6 Conclusion première partie**

Conclusion

- Beaucoup de nouveauté : c'est presque un nouveau langage !
- Possibilité d'avoir une phase d'"exécution" statique durant la compilation grâce aux `constexpr` et `constexpr` ;
- Pas beaucoup de nouveautés sur l'héritage, mais plus sur le fonctionnel, la généréité, etc.