C++ moderne

Sommaire

Thierry Vaira - <tvaira@free.fr> - version v1.6 - 28/12/2020

1. C++

C++ est un langage de programmation compilé permettant la programmation sous de multiples paradigmes (comme la programmation procédurale, **orientée objet** ou générique). Ses bonnes performances, et sa compatibilité avec le C en font un des langages de programmation les plus utilisés dans les applications où la performance est critique.

Créé initialement par **Bjarne Stroustrup** dans les années 1980, le langage C++ est aujourd'hui normalisé par l'ISO. Sa première normalisation date de **1998** (ISO/CEI 14882:1998), ensuite amendée par l'erratum technique de **2003** (ISO/CEI 14882:2003). Une importante mise à jour a été ratifiée et publiée par l'ISO en septembre 2011 sous le nom de ISO/IEC 14882:2011, ou C++11. Depuis, des mises à jour sont publiées régulièrement : en 2014 (ISO/CEI 14882:2014 ou C++14) puis en 2017 (ISO/CEI 14882:2017 ou C++17). [source : wikipedia.org]

Les changements du langage C++ concernent aussi bien le langage initial que la bibliothèque standard.

La bibliothèque standard du C++ (*C++ Standard Library*) est une bibliothèque de classes et de fonctions standardisées selon la norme ISO pour le langage C++. Elle contient aussi la bibliothèque standard du C. Une des principales briques de la bibliothèque standard du C++ est sans aucun doute la **STL** (*Standard Template Library*), à tel point qu'il y a souvent confusion entre les deux.

Dans l'index de popularité des langages TIOBE, le C représente 16,2 % (première place) et le C++ 7,6 % (quatrième place) en novembre 2020.

Liens :

- www.cplusplus.com
- Comité du standard C++

2. C++ moderne

Le C++ moderne est apparu avec la mise à jour C++11.

Les standards C++11, 14, 17 et bientôt 20 apportent de nombreuses fonctionnalités : gestion automatique de la mémoire via des pointeurs intelligents (*Smart Pointers*), déduction de type automatique à la déclaration via `auto`, etc …

L'implémentation des standards C++ va dépendre du compilateur utilisé, de sa version et des options invoquées.

Exemple sous Ubuntu 18.04 :

```
$ g++ --version
g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
```

```
$ man g++
...
g++ [-std=standard] ...
...
-std=
          Determine the language standard. This option is currently only supported
when compiling C or C++.
...
c11
c1x
iso9899:2011
          ISO C11, the 2011 revision of the ISO C standard.  This standard is
substantially completely supported, modulo bugs, floating-point issues (mainly but not
entirely relating to optional C11 features from Annexes F and G) and the optional
Annexes K (Bounds-checking interfaces) and L (Analyzability). The name c1x is
deprecated.
c++11
c++0x
          The 2011 ISO C++ standard plus amendments. The name c++0x is deprecated.
c++14
c++1y
          The 2014 ISO C++ standard plus amendments. The name c++1y is deprecated.
gnu++14
gnu++1y
          GNU dialect of -std=c++14. This is the default for C++ code. The name
gnu++1y is deprecated.
c++1z
          The next revision of the ISO C++ standard, tentatively planned for 2017.
Support is highly experimental, and will almost certainly change in incompatible ways
in future releases.
```

Il est possible de définir le standard utilisé pour le compilateur C++ dans un projet **Qt**. Pour cela, il suffit de l'indiquer dans la variable `CONFIG` de son fichier `.pro` :

```
CONFIG += c++11
```

3. Les exemples

https://github.com/tvaira/cpp-moderne

4. C++11

Lien : C++11

4.1. Les types et les variables

Le langage C++ est dit fortement typé. Chaque variable possède un type.

Une définition est composée de :

- un type pour définir la convention d'interprétation des valeurs possibles
- un objet qui contient en mémoire la valeur d'un type
- une valeur
- une variable qui est le nom de l'objet

Chaque type est directement lié à une architecture matérielle et possède une taille fixe. La taille d'un objet et/ou d'un type est obtenue avec l'opérateur `sizeof`.

L'opérateur `typeid()` (dans type_info) permet lui d'obtenir le type d'une valeur à l'exécution :

```cpp
#include <iostream>
#include <string>
#include <typeinfo>

using namespace std;

class Foo
{
    public:
        Foo(const string& str) : str(str) {}
    private:
        string str;
};

int main()
{
    int i = 10;
    int* pi = &i;
    string s = "hello";

    cout << "i : " << typeid(i).name() << '\n';
    cout << "&i : " << typeid(&i).name() << '\n';
    cout << "pi : " << typeid(pi).name() << '\n';
    cout << "*pi : " << typeid(*pi).name() << '\n';
    cout << "s : " << typeid(s).name() << '\n';

    auto booleen = false; // bool
    auto f = 1.5; // double

    cout << "booleen : " << typeid(booleen).name() << '\n';
    cout << "f : " << typeid(f).name() << '\n';

    Foo foo(s);
    cout << "foo : " << typeid(foo).name() << '\n';

    return 0;
}
```

Il existe un moyen d'insérer des chaînes de caractères complexes dans le code source sans le formatter avec `R"(raw_string)"`. Ceci est pratique avec des chaînes qui contiennent des guillemets `"` et/ou des *antislash* `\`.

Lien : string_literal

```
#include <iostream>

using namespace std;

int main()
{
    string str1 = "<a href=\"file\">C:\\Program Files\\</a>"; // avant
    string str2 = R"(<a href="file">C:\Program Files\</a>)"; // C++11

    cout << "str1 = " << str1 << endl;
    cout << "str2 = " << str2 << endl;

    return 0;
}
```

4.2. Initialisation

Avant d'être utilisé, un objet doit être initialisé. Il existe l'opérateur `=`, les crochets `{}` ou les parenthèses `()` comme initialiseurs universels :

```
int a = 10;
int b(20);
int t[3] = { 1, 2, 3 };
```

Le nouveau standard ISO a introduit une syntaxe d'initialisation uniforme avec les accolades `{}`.

```
int a { 10 };
int b { 20 };
int t[3] { 1, 2, 3 };

std::vector<int> v { 1,2,3,4,5,6,7 };
```

En C++03, il est possible d'assigner une valeur par défaut aux attributs statiques et constantes directement dans le fichier d'en-tête. C++11 étend cette possibilité aux attributs des classes :

```
#include <iostream>

using namespace std;

class X
{
public:
    X() {}
    explicit X(int valeur) : valeur(valeur) {}
    int getValeur() const { return valeur; }

private:
    int valeur = 1; // pour tous les constructeurs
};

int main()
{
    X x1;
    X x2(2);

    cout << "x1 = " << x1.getValeur() << " (" << sizeof(x1) << " octets)" << endl;
    cout << "x2 = " << x2.getValeur() << " (" << sizeof(x2) << " octets)" << endl;

    //cout << "Membre valeur -> " << sizeof(X::valeur) << " octets" << endl; // si
membre public

    return 0;
}
```

4.3. auto

Il est aussi possible de laisser le compilateur déduire le type à la compilation en utilisant le mot-clé `auto` :

```cpp
#include <iostream>

using namespace std;

int main()
{
    auto booleen = false; // bool
    auto ch = 'c'; // char
    auto i = 10; // int
    auto f = 1.5; // double
    auto s = "string"; // char *

    cout << "booleen = " << booleen << " (" << sizeof(booleen) << " octet)" << endl;
    cout << std::boolalpha << "booleen = " << booleen << " (" << sizeof(booleen) << "
octet)" << endl;
    cout << "ch = " << ch << " (" << sizeof(ch) << " octet)" << endl;
    cout << "i = " << i << " (" << sizeof(i) << " octets)" << endl;
    cout << "f = " << f << " (" << sizeof(f) << " octets)" << endl;
    cout << "s = " << s << " (" << sizeof(s) << " octets)" << endl;

    vector<int> v { 1,2,3,4,5,6,7 };

    cout << "v : ";
    for (auto it=v.begin(); it != v.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << '\n';

    // ou avec range-for
    cout << "v : ";
    for (auto i: v) // i est un int
    {
        cout << i << ' ';
    }
    cout << '\n';

    return 0;
}
```

4.4. Membre mutable

Dans une fonction `const`, il est impossible de modifier un attribut (une variable membre) sauf si ce membre est préfixé du mot-clé `mutable`.

Un membre `mutable` n'est jamais `const` !

Lien : mutable specifier

```cpp
#include <iostream>

using namespace std;

class X
{
    public:
        X() : x(0) {}
        int getX() const { return x; };
        void foo() const;

    private:
        mutable int x = 0;
};

void X::foo() const
{
    ++x;
}

int main()
{
    X unObjetX;

    cout << "x = " << unObjetX.getX() << endl;

    unObjetX.foo();

    cout << "x = " << unObjetX.getX() << endl;

    return 0;
}
```

4.5. Les pointeurs

Il faut maintenant utiliser `nullptr` à la place de `0` ou `NULL` pour initialiser un pointeur :

```cpp
#include <iostream>

using namespace std;

int main()
{
    int j = 20;
    int *pj = nullptr;

    cout << "j = " << j << " (" << sizeof(j) << " octets)" << endl;
    cout << "&j = " << &j << " (" << sizeof(&j) << " octets)" << endl;
    cout << "pj = " << pj << " (" << sizeof(pj) << " octets)" << endl;
    if(pj != nullptr)
        cout << "*pj = " << *pj << " (" << sizeof(*pj) << " octets)" << endl;

    pj = &j;
    *pj = 30;
    cout << "j = " << j << " (" << sizeof(j) << " octets)" << endl;
    cout << "&j = " << &j << " (" << sizeof(&j) << " octets)" << endl;
    cout << "pj = " << pj << " (" << sizeof(pj) << " octets)" << endl;
    if(pj)
        cout << "*pj = " << *pj << " (" << sizeof(*pj) << " octets)" << endl;

    return 0;
}
```

4.6. Les pointeurs intelligents

Un pointeur intelligent (*smart pointer*) est un type abstrait de données qui simule le comportement d'un pointeur en y ajoutant des fonctionnalités telles que la libération automatique de la mémoire allouée ou la vérification des bornes.

En C++11, les pointeurs intelligents sont implémentés à l'aide de *templates* qui "imitent" le comportement des pointeurs grâce à la surcharge des opérateurs, tout en fournissant des algorithmes de gestion mémoire.

- `unique_ptr` est une classe qui possède un membre qui pointe sur une ressource (objet) non partageable. `unique_ptr` gère l'objet pointé en devenant responsable de sa suppression lorsqu'il passe hors de portée.

Lien : unique_ptr

```cpp
#include <iostream>
#include <memory>

using namespace std;

class Point
{
    private:
        double x;
        double y;

    public:
        Point() : Point(0., 0.) { }
        Point(double x, double y) : x(x), y(y) { cout << __FUNCTION__ << endl; }
        Point(const Point & p) : x(p.x), y(p.y) { cout << __FUNCTION__ << endl; }
        ~Point() { cout << __FUNCTION__ << endl; }
        double getX() const { return x; }
        double getY() const { return y; }
};

int main()
{
    unique_ptr<Point> p1(new Point(10, 5));

    std::cout << "p1 : " << (p1 ? "not null" : "null") << endl;
    cout << p1->getX() << "," << p1->getY() << endl;

    unique_ptr<Point> p2(move(p1));

    std::cout << "p1 : " << (p1 ? "not null" : "null") << endl;
    std::cout << "p2 : " << (p2 ? "not null" : "null") << endl;
    cout << p2->getX() << "," << p2->getY() << endl;

    unique_ptr<Point> p3;
    std::cout << "p3 : " << (p3 ? "not null" : "null") << endl;

    //p3 = p2; // erreur !

    return 0;
}
```

- Les `shared_ptr` implémentent le comptage de références, ce qui permet de partager l'objet possédé par un `shared_ptr` entre plusieurs `shared_ptr` sans se soucier de comment libérer la mémoire associée. Lorsque le dernier `shared_ptr` est détruit, l'objet pointé est également détruit.

Lien : shared_ptr

```cpp
#include <iostream>
#include <memory>

using namespace std;

class Point
{
    private:
        double x;
        double y;

    public:
        Point() : Point(0., 0.) { }
        Point(double x, double y) : x(x), y(y) { cout << __FUNCTION__ << endl; }
        Point(const Point & p) : x(p.x), y(p.y) { cout << __FUNCTION__ << endl; }
        ~Point() { cout << __FUNCTION__ << endl; }
        double getX() const { return x; }
        double getY() const { return y; }
};

int main()
{
    shared_ptr<Point> p1(new Point(10, 5));

    std::cout << "p1 : " << (p1 ? "not null" : "null") << endl;
    std::cout << "compteur p1 : " << p1.use_count() << endl;
    cout << p1->getX() << "," << p1->getY() << endl;

    shared_ptr<Point> p2(move(p1));

    std::cout << "p1 : " << (p1 ? "not null" : "null") << endl;
    std::cout << "compteur p1 : " << p1.use_count() << endl;
    std::cout << "p2 : " << (p2 ? "not null" : "null") << endl;
    std::cout << "compteur p2 : " << p2.use_count() << endl;
    cout << p2->getX() << "," << p2->getY() << endl;

    shared_ptr<Point> p3;
    std::cout << "p3 : " << (p3 ? "not null" : "null") << endl;
    std::cout << "compteur p3 : " << p3.use_count() << endl;

    p3 = p2;
    std::cout << "p2 : " << (p2 ? "not null" : "null") << endl;
    std::cout << "compteur p2 : " << p2.use_count() << endl;
    std::cout << "p3 : " << (p3 ? "not null" : "null") << endl;
    std::cout << "compteur p3 : " << p3.use_count() << endl;
    cout << p2->getX() << "," << p2->getY() << endl;
    cout << p3->getX() << "," << p3->getY() << endl;

    return 0;
}
```

- Les `weak_ptr` permettent de voir et d'accéder à une ressource (objet) possédée par un `shared_ptr` mais n'ont aucune influence sur la destruction de ce dernier. Ils servent principalement à s'affranchir du problème des références circulaires.

Lien : weak_ptr

```
#include <iostream>
#include <memory>

using namespace std;

class Point
{
    private:
        double x;
        double y;

    public:
        Point() : Point(0., 0.) { }
        Point(double x, double y) : x(x), y(y) { cout << __FUNCTION__ << endl; }
        Point(const Point & p) : x(p.x), y(p.y) { cout << __FUNCTION__ << endl; }
        ~Point() { cout << __FUNCTION__ << endl; }
        double getX() const { return x; }
        double getY() const { return y; }
};

int main()
{
    shared_ptr<Point> p1(new Point(10, 5));

    weak_ptr<Point> wp1;
    weak_ptr<Point> wp2(wp1);
    weak_ptr<Point> wp3(p1);

    cout << "use_count wp1 : " << wp1.use_count() << '\n';
    cout << "use_count wp2 : " << wp2.use_count() << '\n';
    cout << "use_count wp3 : " << wp3.use_count() << '\n';

    return 0;
}
```

4.7. Les énumérations

De manière générale, les énumérations permettent de grouper des ensembles de valeurs dans un type distinct.

Il y a quelques limitations (donc problèmes !) dans l'utilisation du type `enum` :

```
#include <iostream>

using namespace std;

int main()
{
    // Problème n°1 : Deux énumérations ne peuvent pas partager les mêmes noms
    enum Genre { Masculin, Femimin };
    enum GenrePersonne { Masculin, Femimin }; // error: redeclaration of 'Masculin'

    Genre genre = Masculin;
    GenrePersonne genrePersonne = Femimin;

    cout << "genre = " << genre << endl;
    cout << "genrePersonne = " << genrePersonne << endl;

    // Problème n°2 : Aucune variable ne peut avoir un nom déjà utilisé dans une
énumération
    int Masculin = 10; // error: 'int Masculin' redeclared as different kind of symbol

    cout << "Masculin = " << Masculin << endl;

    // Problème n°3 : Les énumérations ne sont pas un type complétement sécurisé
    enum Couleur { Rouge, Vert, Bleu };

    Couleur couleur = Rouge;

    if (genre == couleur) //warning: comparison between 'enum main()::Genre' and 'enum
main()::Couleur'
        cout << "Égal !";

    return 0;
}
```

C++11 a introduit des **classes enum** (appelées énumérations étendues) qui rendent les énumérations fortement typées. L'énumération de classe ne permet pas la conversion implicite en int et ne compare pas non plus les énumérateurs de différentes énumérations.

Lien : enum

Syntaxe :

```
enum class name { enumerator = constexpr , enumerator = constexpr , ... } // constexpr
= 0 par défaut
enum class name : type { enumerator = constexpr , enumerator = constexpr , ... }
enum class name ; // int par défaut
enum class name : type ;
```

Exemple :

```
#include <iostream>

using namespace std;

int main()
{
    enum class Genre { Masculin, Femimin };
    enum class GenrePersonne { Masculin, Femimin };

    Genre genre = Genre::Masculin;
    GenrePersonne genrePersonne = GenrePersonne::Femimin;

    cout << "genre = " << int(genre) << endl;
    cout << "genrePersonne = " << int(genrePersonne) << endl;

    int Masculin = 10;

    cout << "Masculin = " << Masculin << endl;

    /*enum class Couleur { Rouge, Vert, Bleu };

    Couleur couleur = Couleur::Rouge;

    if (genre == couleur) // error: no match for 'operator==' (operand types are
'main()::Genre' and 'main()::Couleur')
        cout << "Égal !";*/

    return 0;
}
```

4.8. decltype

Le mot-clé `decltype`, introduit dans C++11, permet de définir une expression pour exprimer une déclaration de type. `decltype` « retourne » un type.

Lien : decltype

```cpp
#include <iostream>

using namespace std;

struct X
{
   int i;
   double d;
};

int main()
{
    X x;

    decltype(x) y; // le type de y est X
    decltype(x.i) e; // le type de e est int

    return 0;
}
```

`decltype` est notamment intéressant dans l'écriture de bibliothèques génériques à base de templates. Sinon il est fort probable que vous n'ayez pas à vous en servir.

4.9. Les littéraux utilisateur

C++ fournit un certain nombre de littéraux. Les caractères `12.5` sont un littéral qui est résolu par le compilateur comme un type `double`. Avec l'ajout du suffixe `f` (`12.5f`) le compilateur interprétera la valeur comme un type `float`. Les modificateurs de suffixe (comme `U` pour `unsigned` ou `L` pour `long`) pour les littéraux sont fixés par la spécification C++.

À partir de C++11, il est possible de définir ses propres littéraux afin de fournir des suffixes syntaxiques qui améliore la lisibilité et renforce la sécurité des types.

Lien : user_literal

La bibliothèque standard a elle-même défini des littéraux pour `std::complex` et pour les unités dans les opérations de temps dans `std::chrono` :

```
complex<double> n = (2.0 + 3.0i) * 4;

cout << "n = (2 + 3i) x 4" << endl;
cout << "n = " << n << endl;
cout << "partie réelle de n = " << n.real() << endl;
cout << "partie imaginaire de n = " << n.imag() << endl;
cout << endl;

auto recordDuMonde = 2h + 1min + 39s;
cout << "Record du monde du Marathon : 2 h 01 min 39 s (Eliud Kipchoge en 2018)" <<
endl;
cout << "recordDuMonde = " << recordDuMonde.count() << " s" << endl;
```

Liens :

- complex
- chrono

C++ 11 permet donc à l'utilisateur de définir de nouveaux types de modificateurs littéraux qui construiront des objets basés sur la chaîne de caractères que le littéral modifie.

La transformation des littéraux est redéfinie en deux phases distinctes : *raw* (brut) et *cooked* (préparé). Un littéral *raw* est une séquence de caractères d'un type spécifique, tandis que le littéral *cooked* est d'un type distinct. Le littéral `1234`, en tant que littéral *raw*, est la séquence de caractères '1', '2', '3' et '4'. En tant que littéral *cooked*, il s'agit de l'entier `1234`. Le littéral `0xA` est '0', 'x', 'A' soit l'entier `10`.

Liens :

- raw
- cooked

Tous les littéraux définis par l'utilisateur seront des **suffixes**. La définition de littéraux de préfixe n'est pas possible. Tous les suffixes commençant par n'importe quel caractère sauf le trait de soulignement (`) sont réservés par la norme. Ainsi, tous les littéraux définis par l'utilisateur doivent avoir des suffixes commençant par un trait de soulignement (`).

Les littéraux utilisateur sont définis via un opérateur littéral qui se nomme `operator ""`. en.wikipedia.org

Pour les littéraux numériques, le type du littéral est `unsigned long long` pour les littéraux entiers ou `long double` pour les littéraux à virgule flottante. (*Remarque :* il n'est pas nécessaire d'utiliser des types intégraux signés car un littéral avec un préfixe de signe est analysé comme une expression contenant le signe en tant qu'opérateur de préfixe unaire `operator -`, qu'il est possible de surcharger, et le nombre non signé.)

On va définir une classe `Temperature`. Il sera alors possible de définir un littéral pour les degrés Celsius et un autre pour les Fahrenheit. Ensuite, on sera forcé d'exprimer explicitement l'unité de mesure en écrivant par exemple : `auto t1 = 36.5_celsius` ou `auto t2 = 32.0_fahrenheit`.

```
#include <iostream>

using namespace std;

class Temperature
{
    private:
        long double temperature = { 0 }; // en celsius
        explicit Temperature(long double valeur) : temperature(valeur) { }
        friend Temperature operator"" _celsius(long double valeur); // pour une valeur
en virgule flottante
        friend Temperature operator"" _celsius(unsigned long long valeur); // pour une
valeur entière
        friend Temperature operator"" _fahrenheit(long double valeur);
        friend Temperature operator"" _kelvin(long double valeur);

    public:
        constexpr static long double zero_absolu = 273.15; // en celsius

        long double celsius() { return temperature; }
        long double fahrenheit() { return (temperature*9./5.) + 32.; }
        long double kelvin() { return (temperature + Temperature::zero_absolu); }

        Temperature operator+(Temperature t)
        {
            return Temperature(celsius() + t.celsius());
        }
        friend Temperature operator-(Temperature t);
};

Temperature operator"" _celsius(long double valeur) // pour une valeur en virgule
flottante
{
    return Temperature(valeur);
}

Temperature operator"" _celsius(unsigned long long valeur) // pour une valeur entière
{
    return Temperature(double(valeur));
}

Temperature operator"" _fahrenheit(long double valeur)
{
    return Temperature((5./9.) * (valeur - 32.));
}

Temperature operator"" _kelvin(long double valeur)
{
    return Temperature(valeur - Temperature::zero_absolu);
}
```

```
Temperature operator-(Temperature t)
{
    return Temperature((-1.) * t.celsius());
}

int main()
{
    Temperature zeroCelsius = 32._fahrenheit; //Temperature zeroCelsius = 0_celsius;
    cout << "zeroCelsius = " << zeroCelsius.celsius() << "C " << zeroCelsius.kelvin()
<< "K " << zeroCelsius.fahrenheit() << "F " << endl;

    Temperature zeroAbsolu = 0._kelvin;
    cout << "zeroAbsolu = " << zeroAbsolu.celsius() << "C " << zeroAbsolu.kelvin() <<
"K " << zeroAbsolu.fahrenheit() << "F " << endl;

    Temperature t1 = 36.0_celsius + 42.0_celsius;
    cout << "t1 = 36.0_celsius + 42.0_celsius" << endl;
    cout << "t1 = " << t1.celsius() << "C " << t1.kelvin() << "K " << t1.fahrenheit()
<< "F " << endl;

    Temperature t2 = 36.0_celsius + -42.0_celsius;
    cout << "t2 = 36.0_celsius + -42.0_celsius" << endl;
    cout << "t2 = " << t2.celsius() << "C " << t2.kelvin() << "K " << t2.fahrenheit()
<< "F " << endl;

    auto t3 = 36.0_celsius;
    cout << "t3 = " << t3.celsius() << "C " << t3.kelvin() << "K " << t3.fahrenheit()
<< "F " << endl;

    auto t4 = 36_celsius;
    cout << "t4 = " << t4.celsius() << "C " << t4.kelvin() << "K " << t4.fahrenheit()
<< "F " << endl;

    // Evidemment, ceci n'est plus possible :
    //Temperature t5 = 25; // error: conversion from 'int' to non-scalar type
'Temperature' requested
    //Temperature t5 = 25.; // error: conversion from 'double' to non-scalar type
'Temperature' requested
    //Temperature t5 = 36_fahrenheit; // error: unable to find numeric literal
operator 'operator""_fahrenheit' -> il faudrait donc surcharger operator""
_fahrenheit(unsigned long long valeur)

    return 0;
}
```

4.10. Range-for

Introduit en C++11, la boucle **Range-for** exécute une boucle `for` sur une plage de valeurs, telles que tous les éléments d'un conteneur.

Lien : range-for

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

using namespace std;

int main()
{
    string str = "Hello world!";
    for (char c : str)
    {
        cout << c;
    }
    cout << '\n';

    std::vector<int> v = {0, 1, 2, 3, 4, 5};

    for (const auto &i : v) // acces par référence constante
        cout << i << ' ';
    cout << '\n';

    for (auto i : v) // acces par valeur (i est de type int)
        cout << i << ' ';
    cout << '\n';

    int t[] = {0, 1, 2, 3, 4, 5};
    for (auto n : t) // avec un tableau
        cout << n << ' ';
    cout << '\n';

    for (auto p : {2, 4, 6}) // avec des constantes
        cout << p << ' ';
    cout << '\n';

    return 0;
}
```

4.11. Les expressions rationnelles

La bibliothèque C++ standard prend maintenant (en C++11) en charge les **expressions rationnelles** (*Regular Expressions*) avec l'en-tête `<regex>` via une série d'opérations :

- `regex_match` : correspondance exacte avec une expression rationnelle ;
- `regex_search` : recherche correspondance avec une expression rationnelle ;
- `regex_replace` : recherche correspondance avec une expression rationnelle et la remplace ;

Liens :

- regex
- Syntaxe ECMAScript

```
#include <iostream>
#include <regex>

using namespace std;

int main()
{
    string str = "Le code postal de Sarrians est 84260 et 84000 celui d'Avignon.";
    regex reg {R"(\d{5}?)"};
    smatch matches;

    while (regex_search(str, matches, reg))
    {
        for (auto x:matches) std::cout << x << " ";
        cout << endl;
        //cout << matches.suffix().str() << endl;
        str = matches.suffix().str();
    }

    return 0;
}
```

4.12. Délégation du constructeur

En C++03, un constructeur appartenant à une classe ne peut pas appeler un autre constructeur de cette même classe, ce qui peut entraîner de la duplication de code lors de l'initialisation de ses attributs. En permettant au constructeur de déléguer la création d'une instance à un autre constructeur, C++11 apporte donc une solution.

```cpp
#include <iostream>

using namespace std;

class Nombre
{
    public:
        Nombre(int nombre) : nombre(nombre) {}
        Nombre() : Nombre(42) {}
        int getNombre() const { return nombre; }

    private:
        int nombre;
};

int main()
{
    Nombre n1;
    Nombre n2(2);

    cout << "n1 = " << n1.getNombre() << endl;
    cout << "n2 = " << n2.getNombre() << endl;

    return 0;
}
```

4.13. Héritage des constructeurs

En C++03, les constructeurs d'une classe de base ne sont pas hérités par ses classes dérivées. C++11 permet d'hériter explicitement des constructeurs de la classe de base grâce à l'instruction `using` :

```cpp
#include <iostream>

using namespace std;

class Point
{
    public:
        Point(double x, double y) : x(x), y(y) {}
        Point() : x(0.), y(0.) {}
        friend ostream & operator << (ostream & os, const Point & p);

    private:
        double x;
        double y;
};

ostream & operator << (ostream & os, const Point & p)
{
   os << "<" << p.x << "," << p.y << ">";
   return os;
}

class PointCouleur : public Point
{
    public:
        using Point::Point;
        //...

    private:
        unsigned int couleur;
};

int main()
{
    PointCouleur p1;
    PointCouleur p2(2, 2);

    cout << p1 << endl;
    cout << p2 << endl;

    return 0;
}
```

4.14. Liste d'initialiseurs

C++11 introduit le patron de classe `std::initializer_list` qui permet d'initialiser les conteneurs avec une suite de valeurs entre accolades.

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

// ou :
std::vector<int> v {0, 1, 2, 3, 4, 5};

// ou avec une map :
std::map<string,int> m { {"a", 1}, {"b", 2}, {"c", 3}, {"d", 4}, {"e", 5}, {"f", 6} };
```

Lien : initializer_list

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

using namespace std;

template <class T> class MonVecteur
{
    public:
        MonVecteur(initializer_list<T> liste) : v(liste) {}
        void append(std::initializer_list<T> liste)
        {
            v.insert(v.end(), liste.begin(), liste.end());
        }
    //private:
        vector<T> v;
};

int main()
{
    MonVecteur<int> mv = {1, 2, 3, 4, 5};
    mv.append({6, 7, 8});

    std::cout << "mv : ";
    for (auto i: mv.v)
    {
        std::cout << i << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

4.15. constexpr

Le mot clé `constexpr` a été introduit dans C++11 et amélioré en C++14. `constexpr` déclare un objet utilisable dans ce que la norme appelle des expressions constantes.

Comme `const`, `constexpr` peut être utilisé sur des variables mais aussi des fonctions et des

constructeurs.

Lien : constexpr

```
#include <iostream>

using namespace std;

constexpr size_t getTaille()
{
    return 10;
}

constexpr size_t getTaille(int n)
{
    return 10*n;
}

int main()
{
    constexpr float x = 42.0;
    constexpr int N = 5;

    int t1[N] = { 1, 2, 3, 4, 5 }; // 5 x 4
    int t2[getTaille()];  // 10 x 4
    int t3[getTaille(2)];  // 2 x 10 x 4

    cout << "taille t1 = " << sizeof(t1) << " octets" << endl;
    cout << "taille t2 = " << sizeof(t2) << " octets" << endl;
    cout << "taille t3 = " << sizeof(t3) << " octets" << endl;

    return 0;
}
```

4.16. Les nouveaux spécificateurs de classe (override, default, delete, final)

Le spécificateur `default` permet de demander explicitement la génération automatique de la méthode correspondante. On l'utilise par exemple pour le constructeur de copie, le destructeur et l'opérateur de copie :

```
#include <iostream>

using namespace std;

struct Coordonnee
{
    double x;
```

```
    double y;
    Coordonnee() : x(0.), y(0.) {}
    Coordonnee(double x, double y) : x(x), y(y) {}
};

class Point
{
   private:
      Coordonnee coordonnee;

   public:
      // Constructeurs
      Point() {}
      Point(double x, double y) : coordonnee(x, y) {}
      Point(const Point& point) = default; // constructeur de copie

      // Destructeur
      ~Point() = default;

      // Accesseurs et mutateurs
      double getX() const { return coordonnee.x; }
      void setX(double x) { this->coordonnee.x = x; }
      double getY() const { return coordonnee.y; }
      void setY(double y) { this->coordonnee.y = y; }

      // Surcharge
      Point& operator=(const Point& point) = default; // copie
      friend ostream& operator<<(ostream& os, const Point& point);
};

ostream& operator<<(ostream& os, const Point& point)
{
   os << "<" << point.coordonnee.x << "," << point.coordonnee.y << ">";
   return os;
}

int main()
{
    cout << "Les points :" << endl;
    Point p0, p1(4, 0.0), p2(2.5, 2.5);
    cout << "p0 = " << p0 << endl;
    cout << "p1 = " << p1 << endl;
    cout << "p2 = " << p2 << endl;

    cout << "Constructeur de copie : Point p3(p2)" << endl;
    Point p3(p2);
    cout << "p3 = " << p3 << endl;

    cout << "Opérateur de copie : p0 = p3" << endl;
    p0 = p3;
    cout << "p0 = " << p0 << endl;
```

```
    return 0;
}
```

Inversement, le spécificateur `delete` interdira la génération automatique de la méthode correspondante. Utilisé pour un constructeur de copie et l'opérateur de copie, cela rend les objets de cette classe **non copiable** :

```
class Point
{
   private:
      Coordonnee coordonnee;

   public:
      // Constructeurs
      Point() {}
      Point(double x, double y) : coordonnee(x, y) {}
      Point(const Point& point) = delete; // constructeur de copie

      // Destructeur
      ~Point() = default;

      // Accesseurs et mutateurs
      double getX() const { return coordonnee.x; }
      void setX(double x) { this->coordonnee.x = x; }
      double getY() const { return coordonnee.y; }
      void setY(double y) { this->coordonnee.y = y; }

      // Surcharge
      Point& operator=(const Point& point) = delete; // copie
      friend ostream& operator<<(ostream& os, const Point& point);
};
```

On obtient alors les erreurs suivantes :

```
error: use of deleted function ‘Point::Point(const Point&)'
     Point p3(p2);
note: declared here
       Point(const Point& point) = delete;

error: use of deleted function ‘Point& Point::operator=(const Point&)'
     p0 = p3;
note: declared here
       Point& operator=(const Point& point) = delete;
```

Dans la pratique :

- Si un constructeur est déclaré explicitement, aucun constructeur par défaut n'est

automatiquement généré.

- Si un destructeur virtuel est déclaré explicitement, aucun destructeur par défaut n'est automatiquement généré.
- Si un constructeur de déplacement ou un opérateur d'assignation de déplacement est déclaré explicitement :
  - Aucun constructeur de copie n'est généré automatiquement.
  - Aucun opérateur d'assignation de copie n'est généré automatiquement.
- Si un constructeur de copie, un opérateur d'assignation de copie, un constructeur de déplacement, un opérateur d'assignation de mouvement ou un destructeur est déclaré explicitement :
  - Aucun constructeur de déplacement n'est généré automatiquement.
  - Aucun opérateur d'assignation de déplacement n'est généré automatiquement.

De plus, la norme C++11 spécifie les règles supplémentaires suivantes :

- Si un constructeur de copie ou un destructeur est déclaré explicitement, la génération automatique de l'opérateur d'assignation de copie est déconseillée.
- Si un opérateur d'assignation de copie ou un destructeur est déclaré explicitement, la génération automatique du constructeur de copie est déconseillée.

Dans une déclaration ou une définition de méthode, le spécificateur `override` garantit que la fonction membre est virtuelle et remplace une méthode virtuelle d'une classe de base.

```
class A
{
    public:
        virtual void foo(); // une méthode virtuelle
        void bar(); // une méthode "normale" (non virtuelle)
};

class B : public A
{
    public:
        void foo() const override; // Erreur : signature différente
        void foo() override; // Ok : B::foo() remplace A::foo()
        void bar() override; // Erreur : A::bar() n'est pas une méthode virtuelle
};

int main()
{
    // ...
    return 0;
}
```

`override` permet d'énoncer que l'on fait une surcharge et le compilateur en assurera le contrôle ! Il est donc fortement conseillé d'utiliser systèmatiquement `override`.

Inversement, le spécificateur `final` garantit que la méthode est virtuelle et spécifie qu'elle ne peut pas être remplacée par des classes dérivées. Lorsqu'il est utilisé dans une définition de classe, `final` spécifie que cette classe ne peut pas être dérivée.

```
class Base
{
    public:
        virtual void foo(); // une méthode virtuelle
};

class A : public Base // A hérite (est dérivée) de Base
{
    void foo() final; // Ok : Base::foo() est remplacée et A::foo() ne sera pas
remplacée
    void bar() final; // Erreur : A::bar() n'est pas une méthode virtuelle
};

class B final : public A // B hérite (est dérivée) de A et ne sera pas dérivable
{
    void foo() override; // Erreur: foo() n'est pas remplaçable car final dans A
};

class C : public B // Erreur : B est final
{
};

int main()
{
    // ...
    return 0;
}
```

`final` permet de se protéger d'un remplacement non désiré et le compilateur en assurera le contrôle ! Il est donc fortement conseillé d'utiliser systèmatiquement `final`.

4.17. Référence sur rvalue

Chaque expression C++ a un type et appartient à une **catégorie de valeur** (`lvalue`, `rvalue`, ...). Pour rappel, une `lvalue` (*left value* ou valeur à gauche) peut apparaître à gauche d'un opérateur d'affectation (un nom de variable par exemple). Une `rvalue` (*right value* ou valeur à droite) peut apparaître à droite d'un opérateur d'affectation (une expression par exemple). Maintenant, une `rvalue` peut être une `prvalue` (*pure value*) ou `xvalue` (*eXpiring value*).

Une `prvalue` est une expression dont l'évaluation :

- calcule une valeur qui n'est pas associée à un objet
- ou crée un objet temporaire et le désigne

Une `xvalue` est une `glvalue` qui désigne un objet dont les ressources peuvent être réutilisées. Une `glvalue` (*generalized lvalue*) est une expression dont l'évaluation détermine l'identité d'un objet.

```
int a = 2 + 3;
// a est une lvalue
// 2 + 3 est une rvalue
// l'expression 2 + 3 est évaluée à 5 et cette valeur temporaire (et non nommée) est
affectée à la lvalue a

// Idem pour des objets
Point p1, p2; // deux objets Point
Point  p = p1 + p2;
// p est une lvalue
// p1 + p2 est une rvalue
// l'expression p1 + p2 est évaluée (si l'opérateur + est surchargé pour la classe
Point) et un nouvel objet Point est créé temporairement (et non nommé) pour être
affecté à l'objet p (si l'opérateur = est surchargé pour la classe Point)
```

Lien : value_category

Il est possible de créer des références sur des `lvalue` (avec l'opérateur « `&` ») et en C++11 sur des `rvalue` (avec l'opérateur « `&&` ») :

```
void foo(int& x) // ici x est une référence sur une lvalue (et x est une lvalue)
{
    cout << "foo(int x) -> " << x << endl;
}

void foo(int&& x) // ici x est une référence sur une rvalue (et x est une lvalue)
{
    cout << "foo(int&& x) -> " << x << endl;
}

int main()
{
    // Pour rappel :
    int a = 2; // a est une lvalue
    int& ra = a; // ra est une référence sur la lvalue 'a'
    int&& rvb = 42; // rvb est une référence sur une rvalue

    int b = 2; // b est un lvalue

    foo(b); // passage d'une lvalue

    foo(42); // passage d'une rvalue

    return 0;
}
```

On obtient :

```
foo(int x) -> 2
foo(int&& x) -> 42
```

Les références sur `rvalue` prennent en charge l'implémentation de la notion de **déplacement** (ce qui améliorera les performances en évitant des copies inutiles). La notion de déplacement est l'idée de transfèrer les ressources (telles que la mémoire allouée de manière dynamique) d'un objet vers un autre sans avoir à le copier.

4.18. La fonction move()

La fonction `std::move()` retourne une référence `rvalue` sur l'objet passé en argument. Il s'agit d'une fonction de service pour utiliser la notion (ou sémantique) de **déplacement**. La notion de déplacement est l'idée de transfèrer les ressources (telles que la mémoire allouée de manière dynamique) d'un objet vers un autre sans avoir à le copier.

Dans la bibliothèque standard, le déplacement implique que l'objet déplacé est laissé dans un état valide mais non spécifié. Ce qui signifie qu'après une telle opération, la valeur de l'objet déplacé ne doit être que détruite ou affectée d'une nouvelle valeur; y accéder donnera sinon une valeur non spécifiée.

Donc : dans une opération de déplacement (*move*), l'état de l'objet déplacé devient non défini. Cet objet ne doit plus être utilisé. Qu'est-ce que cela veut dire ? Si on déplace un objet p1 dans un objet p2, l'état de p1 n'est plus disponible car il n'est plus défini. Il ne faut donc plus utiliser p1 (mais p1 reste un objet valide). Seul l'objet p2 est viable.

Lien : move

Exemples :

```
int a = 2; // a est une lvalue
int&& rva = std::move(a); // rva est une référence sur une rvalue

Point p1(2.5, 2.5);
Point&& rp1 = std::move(p1);

Point p2(2.5, 2.5);
Point p3(std::move(p2));

void swap(Point& a, Point& b)
{
    Point tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

Point p4(2.5, 2.5), p5(1., 1.);
swap(p4, p5);
```

Il est possible de "convertir" une `lvalue` en référence `rvalue` en utilisant donc la fonction `std::move()` ou `static_cast` :

```
foo(static_cast<int&&>(b)); // la lvalue 'b' est castée en référence rvalue

foo(move(b)); // move() retourne une référence rvalue sur 'b'
```

On obtient :

```
foo(int&& x) -> 2
foo(int&& x) -> 2
```

4.19. Déplacement (constructeur et opérateur)

Le C++11 introduit un nouveau constructeur : le constructeur de déplacement. Sa signature sera : `T(T&& t)`. Son objectif est de "voler" les ressources de l'objet passé en paramètre tout en le laissant dans un état valide mais non spécifié (cet objet passé en paramètre pourra par la suite être détruit ou recevoir une nouvelle valeur). L'objectif du constructeur de déplacement est donc d'éviter des

copies inutiles et par conséquence d'améliorer les performances du programme.

Exemples d'appel de constructeurs :

```
Point p1; // constructeur par défaut

Point p2(p1); // constructeur de copie

Point p3(Point()); // constructeur de déplacement : Point() instancie un objet
temporaire non nommé passé par référence rvalue (inutile de le copier car il suffit de
le "déplacer" en lui volant ses ressources)
```

Le C++11 introduit un nouvel opérateur d'affectation : l'opérateur de déplacement. Sa signature sera : `T& operator=(T&& t)`. Son objectif est de "voler" les ressources de l'objet passé en paramètre tout en le laissant dans un état valide mais non spécifié (cet objet passé en paramètre pourra par la suite être détruit ou recevoir une nouvelle valeur). L'objectif de l'opérateur de déplacement est donc d'éviter des copies inutiles et par conséquence d'améliorer les performances du programme.

Exemples d'appel de l'opérateur d'affectation = :

```
p1 = p2; // opérateur d'affectation = de copie

p3 = p1 + p2; // opérateur d'affectation = de déplacement : (p1 + p2) génére un objet
temporaire non nommé passé par référence rvalue (inutile de le copier car il suffit de
le "déplacer" en lui volant ses ressources)
```

Le constructeur de déplacement et l'opérateur de déplacement utilisent les références sur `rvalue`. On peut leur ajouter le qualificateur `noexcept` s'ils ne lancent pas d'exception.

Liens :

- move_constructor
- move_assignment

Exemple : pour une classe `Point`

```
struct Coordonnee
{
    double x;
    double y;
    Coordonnee() : x(0.), y(0.) {}
    Coordonnee(double x, double y) : x(x), y(y) {}
};

class Point
{
   private:
      Coordonnee *coordonnee;
```

```
   public:
      // Constructeurs
      Point() : coordonnee(new Coordonnee()) { }
      Point(double x, double y) : coordonnee(new Coordonnee(x, y)) { }
      Point(const Point& point); // copie
      Point(Point&& point) noexcept; // déplacement

      // Destructeur
      ~Point() { if(coordonnee) delete coordonnee; };

      // Accesseurs et mutateurs
      double getX() const { return coordonnee->x; }
      void setX(double x) { this->coordonnee->x = x; }
      double getY() const { return coordonnee->y; }
      void setY(double y) { this->coordonnee->y = y; }

      // Surcharge
      Point& operator=(const Point& point); // copie
      Point& operator=(Point&& point); // déplacement
      friend ostream& operator<<(ostream& os, const Point& point);
      friend Point operator+(const Point& p1, const Point& p2);

      // Services (exemples)
      static void swap_v1(Point& a, Point& b);
      static void swap_v2(Point& a, Point& b);
};

// Constructeur de copie
Point::Point(const Point& point) : coordonnee(new Coordonnee(point.coordonnee->x,
point.coordonnee->y))
{
}

// Constructeur de déplacement (le "vol")
Point::Point(Point&& point) noexcept : coordonnee(point.coordonnee)
{
    point.coordonnee = nullptr;
}

// Copie
Point& Point::operator=(const Point& point)
{
    if(this != &point)
    {
        delete coordonnee;
        coordonnee = new Coordonnee(point.coordonnee->x, point.coordonnee->y);
    }
    return *this;
}
```

```
// Déplacement
Point& Point::operator=(Point&& point)
{
    if(this != &point)
    {
        delete coordonnee;
        coordonnee = point.coordonnee; // "vol"
        point.coordonnee = nullptr; // valide mais non spécifié
    }
    return *this;
}

// Surcharge
ostream& operator<<(ostream& os, const Point& point)
{
   os << "<" << point.coordonnee->x << "," << point.coordonnee->y << ">";
   return os;
}

Point operator+(const Point& p1, const Point& p2)
{
   Point p;
   p.coordonnee->x = p1.coordonnee->x + p2.coordonnee->x;
   p.coordonnee->y = p1.coordonnee->y + p2.coordonnee->y;
   return p;
}

void Point::swap_v1(Point& a, Point& b) // par copie
{
    Point tmp(a);   // constructeur de copie
    a = b;          // opérateur de copie
    b = tmp;        // opérateur de copie
}

void Point::swap_v2(Point& a, Point& b) // par déplacement
{
    Point tmp(move(a));
    a = move(b);
    b = move(tmp);
}
```

Exemple n°1 : le déplacement en action

```
cout << "points :" << endl;
Point p2, p3(1.,1.), p4(2.5, 2.5);
cout << "p2 = " << p2 << endl;
cout << "p3 = " << p3 << endl;
cout << "p4 = " << p4 << endl;
cout << endl;

cout << "p2 = p3 + p4" << endl;
p2 = p3 + p4; // move
cout << "p2 = " << p2 << endl;
cout << endl;

cout << "p5 <- p2" << endl;
Point p5(move(p2)); // move
cout << "p5 = " << p5 << endl;
cout << endl;

cout << "p3 <-> p4" << endl;
Point::swap_v1(p3, p4); // par copie
cout << "p3 = " << p3 << endl;
cout << "p4 = " << p4 << endl;
cout << endl;

cout << "p3 <-> p4" << endl;
Point::swap_v2(p3, p4); // move
cout << "p3 = " << p3 << endl;
cout << "p4 = " << p4 << endl;
cout << endl;
```

On obtient :

```
points :
default Point 0x7ffc389b1908
Point 0x7ffc389b1910
Point 0x7ffc389b1918
p2 = <0,0>
p3 = <1,1>
p4 = <2.5,2.5>

p2 = p3 + p4
default Point 0x7ffc389b1920
move operator= 0x7ffc389b1908
p2 = <3.5,3.5>

p5 <- p2
move Point 0x7ffc389b1920
p5 = <3.5,3.5>

p3 <-> p4
copy Point 0x7ffc389b18c0
copy operator= 0x7ffc389b1910
copy operator= 0x7ffc389b1918
p3 = <2.5,2.5>
p4 = <1,1>

p3 <-> p4
move Point 0x7ffc389b18c0
move operator= 0x7ffc389b1910
move operator= 0x7ffc389b1918
p3 = <1,1>
p4 = <2.5,2.5>
```

Exemple n°2 : amélioration des performances

```
auto start = std::chrono::high_resolution_clock::now(); // démarrage chronomètre

vector<Point> points;
for (int i = 0; i < 1000000; ++i)
{
   points.push_back(Point(i, i*2));
}

//vector<Point> courbe(points); // test 1 : par copie

vector<Point> courbe(move(points)); // test 2 : par déplacement

reverse(courbe.begin(), courbe.end()); // pour s'amuser ;)

auto end = chrono::high_resolution_clock::now(); // arrêt chronomètre
chrono::duration<double> elapsed = end - start; // calcul du temps

// Affichage des résultats
cout << "Duration : " << elapsed.count() << " s\n";
cout << "Constructions : " << Point::constructions << "\n";
cout << "Copies : " << Point::copies << "\n";
cout << "Deplacements : " << Point::deplacements << "\n";
cout << "Total : " << (Point::constructions + Point::deplacements) << "\n";
```

On obtient :

- par copie :

```
Duration : 0.193734 s
Constructions : 4548575
Copies : 1000000
Deplacements : 0
Total : 4548575
```

- par déplacement :

```
Duration : 0.0961808 s
Constructions : 1000000
Copies : 0
Deplacements : 3548575
Total : 4548575
```

4.20. Threads

C++11 fournit une classe pour représenter les *threads* d'exécution individuels.

Un *thread* est un fil d'exécution (une séquence d'instructions) qui peut être exécuté simultanément

avec d'autres fils de ce type dans des environnements *multithreading*, tout en partageant un même espace d'adressage.

Un objet thread initialisé représente un *thread* d'exécution actif. Un tel objet thread est joignable et possède un identifiant de *thread* unique.

Lien : thread

Exemple avec un *thread* :

```
#include <iostream>
#include <thread>

// $ g++ thread-1.cpp -lpthread

using namespace std;

void unThread() { cout << "Hello !" << endl; }

int main()
{
    thread hello(unThread); // création et lancement du thread

    hello.join(); // attendre la fin du thread

    return 0;
}
```

Exemple avec deux *threads* :

```cpp
#include <iostream>
#include <thread>
#include <chrono>

// $ g++ thread-2.cpp -lpthread

using namespace std;

void etoile()
{
    for(int i=0; i < 10; ++i)
    {
        this_thread::sleep_for(chrono::duration<int,milli>(250));
        cout << "*";
    }
}

void diese()
{
    for(int i=0; i < 10; ++i)
    {
        this_thread::sleep_for(chrono::duration<int,milli>(250));
        cout << "#";
    }
}

int main()
{
    setbuf(stdout, NULL);

    thread t1(etoile); // création et lancement du thread
    thread t2(diese); // création et lancement du thread

    t1.join(); // attendre la fin du thread
    t2.join(); // attendre la fin du thread

    cout << endl;

    return 0;
}
```

Voir aussi :

- call_once
- atomic

4.21. std::future et std::async

`std::future` est un objet qui peut récupérer une valeur de manière synchronisée. `std::async`permet d'appeler une fonction de manière asynchrone (sans attendre la fin de l'exécution de la fonction). La valeur retournée par la fonction sera accessible via l'objet `future` retourné lors de l'appel et en appelant sa méthode `get()`.

Liens :

- future
- async

```
#include <iostream>
#include <future> // pour async et future
#include <chrono>

// $ g++ future.cpp -lpthread

using namespace std;

// la factorielle d'un entier naturel n est le produit des nombres entiers strictement
positifs inférieurs ou égaux à n
long factorielle(long n)
{
    return n > 1 ? (n * factorielle(n-1)) : 1; //
https://fr.wikipedia.org/wiki/Factorielle#Algorithme
}

// exemple : http://www.cplusplus.com/reference/future/future/
bool is_prime(int x)
{
    // version non optimisée
    for (int i=2; i<x; ++i)
    {
        if (x%i == 0)
            return false;
    }

    return true;
}

int main()
{
    // future permet de lancer une fonction de manière asynchrone et
    // d'en récupérer le résultat
    long n = 15;
    future<long> resultat1 = async(factorielle, n);

    cout << "veuillez patienter pendant le calcul de la factorielle de " << n;
    chrono::milliseconds tempo(100);
```

```
    while (resultat1.wait_for(tempo)==future_status::timeout)
        cout << '.' << flush;
    cout << '\n';

    cout << "résultat : " << resultat1.get() << "\n";

    future<bool> resultat2 = async(is_prime, 444444443);

    cout << "veuillez patienter pendant la vérification";
    while (resultat2.wait_for(tempo)==future_status::timeout)
        cout << '.' << flush;
    cout << '\n';

    cout << "444444443 " << (resultat2.get() ? "est" : "n'est pas") << " premier.\n";

    return 0;
}
```

4.22. Mutex

Un `mutex` est un objet verrouillable conçu pour protéger les accès aux sections critiques de code en empêchant d'autres threads de s'exécuter simultanément et d'accéder aux mêmes emplacements mémoire.

Lien : mutex

```
#include <iostream>
#include <thread>
#include <mutex>

// Chaque thread (tache) va faire ses COUNT boucles
#define COUNT  5000

//#define MUTEX // avec ou sans mutex

using namespace std;

int value_globale = 1;

#ifdef MUTEX
mutex m;
#endif

// Avec mutex : g++ mutex.cpp -DMUTEX -lpthread
// Sans mutex : g++ mutex.cpp -lpthread

void increment()
{
    int value = 0;
```

```
    for(int i=0; i < COUNT; ++i)
    {
        #ifdef MUTEX
        m.lock();
        #endif

        // Récupère la value
        value = value_globale;

        // Incrémente la value
        value += 1;

        // Stocke la value
        value_globale = value;

        #ifdef MUTEX
        m.unlock();
        #endif
    }
}

void decrement()
{
    int value = 0;

    for(int i=0; i < COUNT; ++i)
    {
        #ifdef MUTEX
        m.lock();
        #endif

        // Récupère la value
        value = value_globale;

        // Décrémente la value
        value -= 1;

        // Stocke la value
        value_globale = value;

        #ifdef MUTEX
        m.unlock();
        #endif
    }
}

int main()
{
    setbuf(stdout, NULL);
```

```
    cout << "Avant l'exécution des threads : value = "<< value_globale << " (" <<
COUNT << " boucles)\n";

    thread t1(increment); // création et lancement du thread
    thread t2(decrement); // création et lancement du thread

    t1.join(); // attendre la fin du thread
    t2.join(); // attendre la fin du thread

    cout << "\nApres l'exécution des threads : value = "<< value_globale << " (" <<
COUNT << " boucles)\n";

    return 0;
}
```

Voir aussi : lock_guard

`lock_guard` est un objet qui gère un `mutex` en le gardant toujours verrouillé.

```
mutex m;

void foo()
{
    lock_guard<mutex> lock(m); // création et appel à lock()

    // section critique
    ...

} // destruction et appel à unlock()
```

Voir aussi : condition_variable

4.23. std::ref

La fonction `std::ref` (dans `<functional>`) retourne un objet de type `std::reference_wrapper<T>` qui est en fait une référence sur l'élément.

Lien : std::ref

```cpp
#include <iostream>
#include <functional>
#include <thread>

using namespace std;

// $ g++ ref.cpp -lpthread

void foo(int& data)
{
    data = 42;
}

int main()
{
    int i1 = 100;
    cout << "i1 = " << i1 << endl;

    foo(std::ref(i1));
    cout << "i1 = " << i1 << endl;

    i1 = 100;
    cout << "i1 = " << i1 << endl;
    //std::thread t1(foo, i1); // no works
    std::thread t1(foo, std::ref(i1));  // works

    t1.join();
    cout << "i1 = " << i1 << endl;

    return 0;
}
```

4.24. Les tableaux à taille fixe array

C++11 fournit le nouveau type de tableau `std::array` en tant que conteneur standard (défini dans l'en-tête `<array>`). Contrairement aux autres conteneurs standards, les tableaux `array` ont une taille fixe.

`array` fonctionne de la même manière que les tableaux en C sauf qu'il permet d'être copié (opération relativement coûteuse car c'est une copie de la totalité du bloc de mémoire) et peut s'utiliser explicitement en pointeur.

Lien : array

```cpp
#include <iostream>
#include <array>

#define TAILLE  3
```

```
using namespace std;

int main()
{
    // En C/C++
    cout << "-> En C/C++" << endl;

    int t1[TAILLE] = {10, 20, 30};

    cout << "Elements du tableau t1 (avant) : " << endl;
    for (int i=0; i<TAILLE; ++i)
        cout << t1[i] << " ";
    cout << endl;

    for (int i=0; i<TAILLE; ++i)
        ++t1[i];

    cout << "Elements du tableau t1 (après) : " << endl;
    for (int i=0; i<TAILLE; ++i)
        cout << t1[i] << " ";
    cout << endl;

    cout << "Elements du tableau t1 (après) : " << endl;
    for (int element : t1)
        cout << element << " ";
    cout << endl;

    cout << endl;

    // En C++11
    cout << "-> En C++11" << endl;

    array<int,TAILLE> t2 {10, 20, 30};

    cout << "Elements du tableau t2 (avant) : " << endl;
    for (int i=0; i<t2.size(); ++i)
        cout << t2[i] << " ";
    cout << endl;

    for (int i=0; i<t2.size(); ++i)
        ++t2[i];

    cout << "Elements du tableau t2 (après) : " << endl;
    for (int i=0; i<t2.size(); ++i)
        cout << t2[i] << " ";
    cout << endl;

    cout << "Elements du tableau t2 (après) : " << endl;
    for (int element : t2)
        cout << element << " ";
```

```
    cout << endl;

    // Avec un pointeur
    int *t3 = t2.data(); // data() renvoie un pointeur vers le premier élément du
tableau

    // Dans array, les éléments du tableau sont stockés dans des emplacements mémoire
contigus,
    // le pointeur récupéré (ici t3) peut être utilisé pour accéder à n'importe quel
élément du tableau
    cout << "Elements du tableau t2 (avec un pointeur) : " << endl;
    for (int i=0; i<TAILLE; ++i)
        cout << t3[i] << " "; // cout << *(t3+i) << " ";
    cout << endl;

    // Avec un itérateur
    cout << "Elements du tableau t2 (avec un itérateur) : " << endl;
    //for(array<int,TAILLE>::iterator it = t2.begin(); it != t2.end(); ++it)
    for(auto it = t2.begin(); it != t2.end(); ++it)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

4.25. Les listes simplement chaînée

`forward_list` est l'implémentation d'une liste simplement chaînée accessible seulement par sa tête (`front`).

Lien : forward_list

```
#include <iostream>
#include <forward_list>

using namespace std;

int main()
{
    forward_list<int> liste = {10, 20, 30, 40, 50};

    cout << "liste :\n";
    for (int& element : liste)
        cout << element << " ";
    cout << '\n';

    return 0;
}
```

4.26. Le type Tuple

Un tuple est une collection de dimension fixe d'objets de types différents. Tout type d'objet peut être élément d'un tuple. Cette nouvelle fonctionnalité est implémentée dans le nouvel en-tête `<tuple>` et bénéficie des extensions de C++11.

Lien : tuple

```
#include <iostream>
#include <tuple> // cf. http://www.cplusplus.com/reference/tuple/tuple/

using namespace std;

int main()
{
    typedef tuple<string, string, int, double> tuple_1;
    tuple_1 foo("John", "Smith", 50, 1.87);
    cout << get<0>(foo) << " " << get<1>(foo) << endl;
    cout << "Nb elements du tuple : " << tuple_size<tuple_1>::value << endl;

    tuple<double, double, char> p1(0., 0. ,'A');
    cout << get<2>(p1) << " : " << get<0>(p1) << "," << get<1>(p1) << endl;
    get<2>(p1) = 'B';
    cout << get<2>(p1) << " : " << get<0>(p1) << "," << get<1>(p1) << endl;

    auto bar = std::make_tuple("pi", 3.14);
    cout << get<0>(bar) << " = " << get<1>(bar) << endl;

    return 0;
}
```

4.27. Tables de hachage

Une table de hachage (*hash table*) est une structure de données qui permet une association clé-élément. Il s'agit d'un tableau ne comportant pas d'ordre (contrairement à un tableau ordinaire qui est indexé par des entiers). On accède à chaque élément de la table par sa clé. L'accès s'effectue par une fonction de hachage qui transforme une clé en une valeur de hachage (un nombre) indexant les éléments de la table.

Pour éviter les conflits de noms avec les bibliothèques non standards qui ont leur propre implémentation des tables de hachage, on utilisera le préfixe `unordered` au lieu de `hash`.

Il existe deux types de tables de hachage dans la STL :

- `hash_set<K>` : table de hachage simple, stocke seulement des clés de type K.
- `hash_map<K,T>` : table de hachage double, stocke des clés de type K associées à des valeurs de type T. À une clé donnée ne peut être stockée qu'une seule valeur.

Liens :

- hash_set
- hash_map

`hash_set` et `hash_map` font partie de la STL mais ne sont pas intégrés à la bibliothèque standard C++. Les compilateurs GNU C++ et Visual C++ de Microsoft les ont quand même implémentés.

Le standard C++11 propose des conteneurs similaires : `unordered_set` et `unordered_map`.

Liens :

- unordered_set
- unordered_map

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

int main()
{
    unordered_map<string, string> hashtable;

    //hashtable.emplace("www.wikipedia.fr", "78.109.84.114");
    //cout << "Adresse IP : " << hashtable["www.wikipedia.fr"] << endl;

    hashtable.insert(make_pair("www.cplusplus.com", "167.114.170.15"));
    hashtable.insert(make_pair("www.google.fr", "216.58.204.67"));
    cout << "Adresse IP de www.google.fr : " << hashtable["www.google.fr"] << endl <<
endl;

    cout << "La  table : " << endl;
    for (auto itr = hashtable.begin(); itr != hashtable.end(); itr++)
    {
        cout << (*itr).first << " -> " << (*itr).second << endl;
    }

    return 0;
}
```

On peut créer sa propre fonction de hachage avec un foncteur (*Function Object*) est un objet qui se comporte comme une fonction en surchargeant l'opérateur `()` :

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

// Foncteur de hachage
class Hachage
{
    public:
        size_t operator()(const string &s) const
        {
            cout << "[hash : " << hash<string>()(s) << "]" << endl;
            return hash<string>()(s);
        }
};

int main()
{
    unordered_map<string, string, Hachage> hashtable;

    hashtable.insert(make_pair("www.wikipedia.fr", "78.109.84.114"));
    hashtable.insert(make_pair("www.cplusplus.com", "167.114.170.15"));
    hashtable.insert(make_pair("www.google.fr", "216.58.204.67"));

    cout << endl << "La  table : " << endl;
    for (auto itr = hashtable.begin(); itr != hashtable.end(); itr++)
    {
        cout << (*itr).first << " -> " << (*itr).second << endl;
    }
    cout << endl;

    cout << "Adresse IP de www.google.fr : " << hashtable["www.google.fr"] << endl;

    return 0;
}
```

On peut utiliser `unordered_map` avec ses propres classes à condition de définir l'opérateur `==` :

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

class Fabricant
{
    private:
        string nom;
```

```
    public:
        Fabricant(string nom)
        {
            this->nom = nom;
        }

        string getNom() const
        {
            return nom;
        }

        bool operator==(const Fabricant &f) const
        {
            return nom == f.nom;
        }
};

class Modele
{
    private:
        string nom;
        int annee;

    public:
        Modele(string nom, int annee)
        {
            this->nom = nom;
            this->annee = annee;
        }

        string getNom() const
        {
            return nom;
        }

        int getAnnee() const
        {
            return annee;
        }

        bool operator==(const Modele &m) const
        {
            return (nom == m.nom && annee == m.annee);
        }
};

class Hachage
{
    public:
        size_t operator()(const Modele &m) const
```

```
        {
            return hash<string>()(m.getNom()) ^ hash<int>()(m.getAnnee());
        }
};

int main()
{
    unordered_map<Modele, Fabricant, Hachage> catalogue;

    Modele zoe("Zoe", 2012);
    Modele megane3("Megane III", 2008);
    Modele clio3("Clio III", 2005);
    Modele bipper("Bipper", 2007);
    Fabricant renault("Renault");
    Fabricant peugeot("Peugeot");

    catalogue.insert(make_pair(zoe, renault));
    catalogue.insert(make_pair(megane3, renault));
    catalogue.insert(make_pair(clio3, renault));
    catalogue.insert(make_pair(bipper, peugeot));

    for (auto &itr : catalogue)
    {
        cout << itr.second.getNom() << " " << itr.first.getNom() << " " << itr.first
.getAnnee() << endl;
    }

    return 0;
}
```

4.28. Nombres pseudo-aléatoires

La bibliothèque standard du C permet de générer des nombres pseudo-aléatoires grâce à la fonction `rand()`.

C++11 va fournir une manière différente de générer les nombres pseudo-aléatoires :

- un moteur de génération, qui contient l'état du générateur et produit les nombres pseudo-aléatoires ;
- une distribution, qui détermine les valeurs que le résultat peut prendre ainsi que sa loi de probabilité.

C++11 définit trois algorithmes de génération (linear_congruential, subtract with carry et mersenne_twister), chacun ayant des avantages et des inconvénients et fournira un certain nombre de lois standard (uniform_int_distribution, bernoulli_distribution, ...).

Lien : random

```cpp
#include <iostream>
#include <random>
#include <functional>   // std::bind
#include <chrono>

using namespace std;

int main()
{
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();

    std::uniform_int_distribution<int> distribution1(1, 6); // un dé à 6 faces
    std::default_random_engine default_engine(seed);

    int de = distribution1(default_engine);  // genere un nombre entre 1 et 6
    cout << "de = " << de << '\n';

    std::uniform_int_distribution<int> distribution2(0, 99);
    std::mt19937 engine(seed);
    auto generator = std::bind(distribution2, engine);

    int random = generator();  // genere un nombre entre 0 et 99
    cout << "random = " << random << '\n';

    return 0;
}
```

4.29. Fonction lambda

Une lambda est une fonction possiblement anonyme et destinée à être utilisée localement.

Liens :

- lambda
- Des fonctions somme toute lambdas

Syntaxe :

```
[zone de capture](paramètres de la lambda) -> type de retour { instructions }
```

Exemple simpliste :

```
int main()
{
    []() -> void {};

    return 0;
}
```

Exemples basiques :

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

using namespace std;

int main()
{
    // Exemple 1
    auto lambda = [](string const & message) -> void { cout << "Message reçu : " <<
message << endl; };

    lambda("Hello !");

    // Exemple 2
    vector<string> const chaines { "Un mot", "Autre chose", "Du blabla", "Du texe",
"Des lettres" };

    for_each(begin(chaines), end(chaines), [](string const & message) -> void
    {
        cout << "Message reçu : " << message << endl;
    });

    return 0;
}
```

Les expressions Lambda (ou *closure*) sont donc un bon moyen de passer du code en paramètre d'une fonction :

```
// Exemple 3
vector<int> v { 1,2,3,4,5,6,7 };

cout << "v : ";
for (auto i: v) // i est un int
{
    cout << i << ' ';
}
cout << '\n';

unsigned int nbElementsPairs = 0;

nbElementsPairs = count_if(v.begin(), v.end(), [](auto x) { return !(x % 2); });
cout << "nbElementsPairs : " << nbElementsPairs << endl;
```

Lien : count_if

Cette utilisation est simple et place le code du prédicat au bon endroit (sans avoir besoin de déclarer une fonction pour cela).

Les lambdas peuvent accéder aux variables dans la portée par référence ou par valeur :

```
// Exemple 4
int borneMax = 5;
unsigned int nbElements = 0;

// accès aux variables par référence [&]
nbElements = count_if(v.begin(), v.end(), [&](auto x) { return (x <= borneMax); });
cout << "nbElements : " << nbElements << endl;

borneMax = 4;

// accès aux variables par copie [=]
nbElements = count_if(v.begin(), v.end(), [=](auto x) { return (x <= borneMax); });
cout << "nbElements : " << nbElements << endl;
```

4.30. std::function et std::mem_fn

Le template `std::function` permet d'encapsuler un pointeur de fonction ou une lambda :

```cpp
#include <iostream>
#include <string>
#include <functional>

using namespace std;

void foo(string str)
{
    cout << "message : " << str << endl;
}

int main()
{
    std::function<void(string)> fn_foo = foo;

    fn_foo("Hello world!");

    return 0;
}
```

Le template `std::mem_fn` permet d'encapsuler un pointeur de méthode (fonction membre) d'une classe :

```cpp
#include <iostream>
#include <string>
#include <functional>

using namespace std;

class Foo
{
    public:
        Foo(const string& str) : str(str) {}
        void print(int n=1) const { for(int i = 0; i<n; ++i) cout << str << '\n'; }
    private:
        string str;
};

int main()
{
    const Foo foo("Hello world!");

    auto fn1 = mem_fn(&Foo::print);
    fn1(foo, 5);

    return 0;
}
```

Liens :

- std::function
- std::mem_fn

5. C++14

C++14 a été une évolution mineuyre de la norme.

Lien : C++14

5.1. Nombres binaires

Avec le C++14, il est désormais possible de spécifier des nombres binaires en utilisant le préfixe `0b` ou `0B` :

```
#include <iostream>

using namespace std;

int main()
{
    int i = 0b01010101;
    int j = 0B10101010;

    cout << "i = " << i << " (0x" << hex << i << ")" << endl;
    cout << "j = " << j << " (0x" << hex << j << ")" << endl;

    return 0;
}
```

5.2. Séparateur de chiffres

Pour améliorer la lisibilité :

```cpp
#include <iostream>

using namespace std;

int main()
{
    int i = 0b0101'0101;
    int j = 0b1010'1010;

    cout << "i = " << i << " (0x" << hex << i << ")" << endl;
    cout << "j = " << j << " (0x" << hex << j << ")" << endl;

    int un_milliard = 1'000'000'000;

    return 0;
}
```

6. C++17

Liens :

- De C++14 à C++17
- C++17
- C++17 (en)

6.1. Le type byte

En C++17, `std::byte` (dans `<cstddef>`) représente un octet en mémoire. En C/C++, on utilisait le type `char` ou `unsigned char`. Attention, `std::byte` n'est pas un type caractère et ni un type arithmétique. Seuls opérateurs au niveau du bit ont été surchargés :

- les opérateurs de décalage comme `<<`, `>>`, `<<=`, `>>=`
- les opérateurs logiques comme `|`, `&`, `^`, `~`, `|=`, `&=`, `^=`

Le type `byte` n'est pas directement utilisable comme un entier sauf via la fonction `std::to_integer<>()`.

Lien : std::byte

```cpp
#include <iostream>
#include <cstddef>

using namespace std;

// g++ -std=c++17 byte.cpp

int main()
{
    byte b1{ 10 };
    byte b2{ 21 };
    cout << "b1 = " << to_integer<int>(b1) << endl;
    b2 <<= 1;
    cout << "b2 = " << to_integer<int>(b2) << endl;

    byte b3 = b1 & b2;
    cout << "b3 = b1 & b2" << endl;
    cout << "b3 = " << to_integer<int>(b3) << endl;

    int i1 = to_integer<int>(b3);
    cout << "i1 = " << i1 << endl;

    return 0;
}
```

6.2. std::invoke

En C++17, la fonction `std::invoke` (dans `<functional>`) permet d'appeler une fonction ou une méthode en lui passant des arguments.

Lien : std::invoke

```cpp
#include <iostream>
#include <string>
#include <functional>

//$ g++ -std=c++17 invoke.cpp

using namespace std;

class Foo
{
    public:
        Foo(const string& str) : str(str) {}
        void print(int n=1) const { for(int i = 0; i<n; ++i) cout << str << '\n'; }
    private:
        string str;
};

void print(string str)
{
    cout << "message : " << str << endl;
}

int main()
{
    // fonction
    invoke(print, "Hello world!");

    const Foo foo("Hello world!");

    // méthode
    invoke(&Foo::print, foo, 2);

    return 0;
}
```

6.3. std::optional

En C++17, le type `std::optional<T>` peut contenir une valeur ou pas. On utilise la méthode `has_value()` pour déterminer s'il y a une valeur dans l'objet et `value()` pour la récupérer.

Lien : std::optional

```cpp
#include <iostream>
#include <string>
#include <optional>

// $ g++ -std=c++17 optional.cpp
```

```cpp
using namespace std;

class Article
{
    public:
        Article(const string& libelle) : libelle(libelle) {}
        Article(const string& libelle, optional<double> prix) : libelle(libelle),
prix(prix) {}
        string getLibelle() const { return libelle; }
        optional<double> getPrix() const {
            if(prix.has_value())
                return prix;
            else
                return {};
        }
    private:
        string libelle;
        optional<double> prix;
};

int main()
{
    Article article1("De'Longhi Magnifica S", 295.0);
    Article article2("Philips EP2220");

    cout << "Article " << article1.getLibelle() << endl;
    auto prix1 = article1.getPrix();
    if (prix1.has_value())
    {
        cout << "  prix : " << prix1.value() << " euros" << endl;
    }
    else
    {
        cout << "  pas de prix pour cet article" << endl;
    }

    cout << "Article " << article2.getLibelle() << endl;
    auto prix2 = article2.getPrix();
    if (prix2.has_value())
    {
        cout << "  prix : " << prix2.value() << " euros" << endl;
    }
    else
    {
        cout << "  pas de prix pour cet article" << endl;
    }

    return 0;
}
```

6.4. std::any

En C++17, le type `std::any` peut contenir n'importe quel type ou aucune valeur. C'est l'équivalent d'un `void *` *type-safe*. On peut utiliser la méthode `has_value()` pour déterminer s'il y a une valeur. Il existe aussi `any_cast<T>()` pour réaliser des conversions vers des types `T`. `type()` retourne une référence sur le type `type_info`.

Lien : std::any

```
#include <iostream>
#include <string>
#include <any>

// $ g++ -std=c++17 any.cpp

using namespace std;

int main()
{
    any a = 1;
    cout << a.type().name() << " -> " << any_cast<int>(a) << '\n';

    any s = string("Hello world!");
    cout << s.type().name() << " -> " << any_cast<string>(s) << '\n';

    return 0;
}
```

7. C++20

Liens :

- C++20
- C++20 (en)

8. Wikipédia

- C++11
- C++14
- C++17
- C++20

9. Voir aussi

- C++ - Sujets divers

Site : tvaira.free.fr

Thierry Vaira - <tvaira@free.fr> - version v1.6 - 28/12/2020