C++1y - Les tableaux

Pour commencer le tour des nouvelles fonctionnalités proposées par les prochaines normes du C++ (je parle au pluriel puisqu'il devrait y avoir la sortie d'une nouvelle norme, le C++14, et de plusieurs *Technical Specification*. Voir l'article Les derniers jours du C++11 ? pour les détails), je vais faire le tour des différents TS avant d'aborder les détails du C++14. Enfin, je terminerais par faire une review des différents drafts pour le meeting de Issaquah. Dans cet article, je vais donc parler des nouvelles fonctionnalités concernant les tableaux :

- N3820, qui devrait sortir sous forme de TS cette année (ce TS devait initialement faire partie du C++14). Ce TS ajoute le support des tableaux de taille connue à l'exécution (en particulier std::dynarray) ;
- N3824, qui propose l'ajout d'une fonction *make_array* pour créer des tableaux statiques std::array ;
- N3869 et N3870, pour étendre l'utilisation de std::shared_ptr et std::make_shared aux tableaux style C ;
- N3851 et la mise à jour N3976, qui propose l'ajout des tableaux à plusieurs dimensions.

Les tableaux de taille connue à l'exécution

Pour créer un tableau d'éléments contiguë en mémoire, nous avons actuellement plusieurs approches possibles. Les critères permettant d'orienter le choix sont :

- l'utilisation de la bibliothèque standard ou l'utilisation de tableaux style C ;
- la création de tableaux statiques (taille définie à la compilation) ou dynamique (taille définie à l'exécution).

Selon ces 2 critères, nous allons donc avoir 4 choix possibles :

1. Utiliser les tableaux statiques style C

```
int t2[] = {0, 1, 2, 3}; // tableau de 4 entiers
auto s = "un chaine"; // tableau de caractères constants
(const char)
```

Ce type de tableau ne présente pas l'inconvénient de la gestion manuelle de la mémoire, contrairement aux tableaux dynamiques style C que nous verrons ensuite. Ils sont utilisables avec les algorithmes de la bibliothèque standard, mais avec une syntaxe particulière :

```
std::find(t1, t1+4, 2);
```

L'inconvénient est que le tableau n'encapsule pas l'information sur sa taille et donc le risque de se tromper et sortir des limites du tableau est plus grand. De plus, ce code n'est pas évolutif, si on change la taille du tableau, il faut modifier toutes les lignes de code utilisant ce tableau.

2. Utiliser les tableaux statiques std::array

Ce type de tableau a été introduit dans le C++11 pour fournir des tableaux statiques similaires aux tableaux statiques style C, mais avec une interface proche de std::vector. En particulier, la fonction size pour connaitre la taille du tableau ou les fonctions begin et end pour manipuler des itérateurs :

```
std::array<int, 4> a1 = {0, 1, 2, 3}; // tableau d'entier
std::find(begin(a1), end(t1), 2); // itérateurs
```

3. Utiliser les tableaux dynamiques style C

Ces tableaux sont l'un des problèmes majeurs du C++ old-school, que l'on retrouve dans beaucoup de tutoriels sur internet et dans les livres

(même certains livres récents). On voit également beaucoup de débutants sur les forums utiliser ce type de syntaxe, sans maîtriser les problématiques que cela pose. La sanction est généralement simple : le programme plante, avec des messages d'erreurs qui n'indiquent pas correctement la source du problème (je parle souvent d'erreur cryptique. On est tous déjà eu ce type d'erreur, qui prend parfois des heures à déboguer, pour un petit oubli dans une ligne de code).

```
auto v1 = new int [10];
delete[] v1;
```

Ce code pose aussi souvent des problèmes de fuite mémoire (oublier d'appeler delete[] ou appelle de delete), les accès hors de limites du tableau et la taille du tableau ne sont pas gérés, l'utilisation des algorithmes de la bibliothèque standard est moins naturelle.

4. Utiliser les tableaux dynamiques std::vector

Ces tableaux ont été ajoutés dans le C++ depuis très longtemps. Lors de leur ajout, les différentes implémentations de std::vector n'étaient pas toujours performantes, ce qui a justifié à l'époque que beaucoup continuaient d'utiliser les tableaux dynamiques style C. De nos jours, cet argument n'est plus du tout valable, l'utilisation de std::vector devrait être le comportement par défaut des débutants et moins débutants.

```
std::vector<int> v1; // tableau vide
std::vector<int> v2(4); // tableau contenant {0, 0, 0, 0}
std::vector<int> v3(4, 2); // tableau contenant {2, 2, 2, 2}
std::vector<int> v4 = {1, 2, 3, 4}; // tableau {1, 2, 3, 4}

std::find(begin(v4), end(v4), 2); // recherche

for (auto i: v4) // range-based for
    std::cout << i << std::endl;
```

Cette classe présente tous les avantages des conteneurs de la bibliothèque standard : expressivité, sécurité du code (libération de la

mémoire, vérification des accès hors limite), évolutivité du core. Remarque : pour les accès aléatoires dans un tableau, il est préférable d'utiliser par défaut la fonction at, qui vérifie les accès en dehors des limites du tableau, au lieu de l'opérateur [], qui ne le fait pas (en pratique, certain compilateur vérifie les limites avec l'opérateur [], mais cela n'est pas garantie par la norme).

Pourquoi ne pas utiliser les syntaxes style C ?

Ceux qui utilisent les tableaux style C et qui recommandent leur utilisation essaient de transposer les pratiques du C (bonnes ou mauvaises, je ne saurais juger, je ne suis pas assez expert en C) en C++. En faisant cela, ils font une erreur fondamentale : ils pensent que le C++, c'est une évolution du C. Ce qui est vrai historiquement, mais faux en termes de syntaxe : le C et le C++ sont bien deux langages différents, avec leurs problématiques et modes de conceptions différents. Et un code parfaitement valable en C peut être très problématique en C++.

```
class A {
    int* v1;
    int* v2;
public:
    A(unsigned n) {
        if (n == 0) return;
        v1 = new int[n];
        v2 = new int[n]; // problème !
    }
    ~A() {
        delete[] v1;
        delete[] v2;
    }
};
```

Ce code, malgré sa simplicité et son apparente innocence est à bannir en C++ ! Il présente un risque de fuite mémoire important (si vous ne savez pas pourquoi, cela justifie encore plus de ne pas utiliser cette syntaxe), il peut fonctionner quelques temps, puis planter du jour au lendemain. La

localisation du problème peut être longue, c'est une perte de temps inutile (que beaucoup ont déjà malheureusement expérimenté - dont moi 
). La syntaxe correcte est plus simple et plus sûre :

```
class A {
    std::vector<int> v1;
    std::vector<int> v2;
public:
    A(unsigned n) : v1(n), v2(n) {}
};
```

Ce sont ces problèmes qui justifient la règle suivante : en C++ moderne, il faut bannir l'utilisation des pointeurs nus, de new et de delete. (Comme toutes les règles, celle-ci peut être enfreint, mais uniquement lorsque l'on connait les risques et limites de ces syntaxes et que l'on a une très bonne raison de le faire).

Utiliser les tableaux de taille fixée à l'exécution (runtime-sized arrays)

Il reste un dernier cas d'utilisation que je n'ai pas évoqué. Lorsque l'on souhaite avoir un tableau de taille constante, mais non connue à la compilation, on peut utiliser ce type de syntaxe avec les tableaux statiques style C :

```
void foo(unsigned n) {
    int v[n];
}
foo(4);
```

Cela permet de créer un tableau alloué dans la pile, avec une taille déterminée lors de l'appel de la fonction. Cette syntaxe n'est théoriquement pas légale en C++11, mais beaucoup de compilateur l'accepte (pour GCC, il faut utiliser l'option de compilation `-pedantic` pour avoir un avertissement : `warning: ISO C++ forbids variable length array 'v' [-Wvla]`). La première partie du TS ajoute le support

de ce type de tableau en C++, en modifiant le type d'expression acceptée pour créer un tableau :

```
type [constante] label; // C++03
type [constexpr] label; // C++11, avec expression constante
type [expression] label; // TS array
```

La seconde partie du TS ajoute une nouvelle classe dans la bibliothèque standard : std::dynarray, qui permet de créer un tableau de taille fixée à l'exécution et offrant une syntaxe similaire aux autres conteneurs de la bibliothèque standard (size, begin, end, etc.). Les itérateurs sont de type à accès aléatoire (random access iterators), il est possible d'utiliser les opérateurs ++, +=, –, -= et [].

```
// header
#include <dynarray>

// constructeurs
std::dynarray<int> v1(4); // contient {0, 0, 0, 0}
std::dynarray<int> v2(4, 2); // contient {2, 2, 2, 2}
std::dynarray<int> v3 = {1, 2, 3, 4};
auto v4 = v1; // ou auto v4 {v1};

// limites
auto s = size(v1); // nombre d'éléments
bool b1 = empty(v1); // conteneur vide ?

// accès
auto i1 = v1.at(2); // vérification limites
auto i2 = v1[2]; // sans vérifications limites
auto i3 = v1.front(); // premier élément
auto i4 = v1.back(); // dernier élément

// itérateurs
auto it1 = begin(v1); // ou v1.begin()
auto it2 = end(v1); // ou v2.end()
// également cbegin, cend, rbegin, rend, crbegin et crend
// avec c = const, r = reverse

// remplissage
v1.fill(5); // contient {5, 5, 5, 5}
```

```
// opérateurs de comparaison
bool b2 = (v1 == v2);
bool b3 = (v1 != v2);
bool b4 = (v1 > v2); // ordre lexicographique
bool b5 = (v1 >= v2); // ordre lexicographique
bool b6 = (v1 < v2); // ordre lexicographique
bool b7 = (v1 <= v2); // ordre lexicographique
```

Un point important à noter lors de l'utilisation de ce type de tableau. Ces tableaux sont alloués sur la pile, qui a une taille limitée (différente selon le système et les options de compilation). Lorsque l'on souhaite créer des tableaux de taille importante, il faut utiliser des tableaux dynamiques std::vector. Lors de la création d'un tableau de taille fixée à l'exécution, si l'allocation échoue à cause d'un manque de mémoire disponible, le comportement est indéterminé (undefined behavior). En pratique, cela signifie que le comportement est laissé à l'appréciation des concepteurs de compilateurs (et donc potentiellement non reproductible selon le compilateur), bien qu'il est encouragé à lancer une exception de type std::bad_array_length (nouveau type d'exception également ajoutée dans ce TS). La seule garantie est que la mémoire sera libérée correctement (local object with automatic storage duration). Remarque : pour le constructeur des conteneurs avec un initializer-list, il est possible de ne pas mettre le signe égal. Il faut faire attention dans ce cas à ne pas confondre les deux syntaxes, avec parenthèses et avec crochets, qui sont visuellement proches, mais ont des comportements totalement différents :

```
std::dynarray<int> v5 {3, 4}; // contient {3, 4}
std::dynarray<int> v6 (3, 4); // contient {4, 4, 4}
```

La fonction make_array

N3824 est le seul draft du Working Group - Library Evolution pour le meeting de février (les autres travaux de ce WG sont déjà acceptés dans un TS ou dans le C++14). Il propose d'ajouter une fonction make_array, similaire aux fonctions make existantes (make_pair, make_tuple, make_shared et make_unique) pour créer un tableau statique

(std::array). Les deux syntaxes suivantes seront donc possibles :

```
std::array<int, 4> a1 = {1, 2, 3, 4};
auto a2 = make_array(1, 2, 3, 4);
auto a3 = make_array<int>(1, 2, 3, 4); // explicite
```

La principale difficulté de cette syntaxe est de bien comprendre la déduction des types des arguments, pour connaître le type de tableau créé. L'idéal est d'utiliser des types littéraux simples ou de spécifier explicitement le type. Pour rappel des littérales de base du C++ :

```
// entiers
auto a4 = make_array(1, 2, 3, 4); // int
auto a4 = make_array(1L, 2L, 3L, 4L); // long
auto a4 = make_array(1LL, 2LL, 3LL, 4LL); // long long
(C++11)
auto a4 = make_array(1U, 2U, 3U, 4UU); // unsigned
auto a4 = make_array(1UL, 2UL, 3UL, 4UL); // unsigned long
auto a4 = make_array(1ULL, 2ULL, 3ULL, 4ULL); // unsigned
long
                                                // long (C++11)

// réels
auto a4 = make_array(1.0, 2.0, 3.0); // double
auto a4 = make_array(1.0f, 2.0f, 3.0f, 4.0f); // float
auto a4 = make_array(1.0L, 2.0L, 3.0L, 4.0L); // long double
```

Remarque : les suffixes sont insensibles à la casse, il est donc possible d'utiliser u ou U, l ou L, etc. Pour des raisons de risque de confusion entre l (L minuscule) et 1 (un) avec certaines polices de caractères, je préfère l'utilisation de L en majuscule. Remarque 2 : il n'est pas obligatoire d'écrire le chiffre 0 pour les nombres réels lorsque celui-ci est le seul chiffre après le séparateur décimal (1.0 est équivalent à 1.) Pour des raisons de lisibilité, je préfère mettre le 0.

```
// caractères
auto a4 = make_array('a', 'b'); // char
auto a4 = make_array(L'a', L'b'); // wchar_t
auto a4 = make_array(u'a', u'b'); // char16_t (C++11)
auto a4 = make_array(U'a', U'b'); // char32_t (C++11)
```

```
auto a4 = make_array('a+', 'b+'); // int

// chaîne
auto a4 = make_array("a", "b"); // const char[]
auto a4 = make_array(L"a", L"b"); // const wchar_t[]
auto a4 = make_array(u8"a", u8"b"); // const char[] (C++11)
auto a4 = make_array(u"a", u"b"); // const char16_t[] (C++11)
auto a4 = make_array(U"a", U"b"); // const char32_t[] (C++11)
```

Remarque : le C++11 ajoute également les chaînes littérales brutes (Raw String Literal), qui ne prennent pas en compte les caractères d'échappement. Par exemple : R"\t\n" ne correspond pas à 2 caractères (tabulation suivie par un retour à la ligne), mais à 4 caractères ('\', 't', '\', 'n'). Ces chaînes brutes peuvent être précédées par les autres préfixes, qui modifient le type en conséquence (ainsi, UR"a" est une chaîne brute de type const char32_t[]). Remarque 2 : le C++14 prévoit l'ajout des chaînes littérales utilisateurs dans la bibliothèque standard, par exemple pour ajouter le suffixe "s" pour créer des chaînes de type std::string, utilisable avec la déduction automatique des types :

```
auto s = "a"s; // std::string
auto a4 = make_array("a"s, "b"s); // std::string
```

Il peut être intéressant de pouvoir créer un tableau à partir d'un autre tableau style C. Par exemple :

```
int a1[] = {1, 2, 3, 4};
auto a2 = make_array(a1); // crée un tableau de 1 élément
                          // de type int[]
```

En complément de make_array, le draft propose également la fonction to_array, qui prend en paramètre un tableau style C :

```
auto a3 = to_array(a1); // contient {1, 2, 3, 4}
auto a4 = to_array("abcd"); // contient {'a', 'b', 'c', 'd'}
```

La déduction des types est réalisée après suppression des const et volatile (grâce à remove_cv) et des références (grâce à remove_reference). Il est cependant possible de créer des tableaux de références en utilisant std::reference_wrapper ou std::ref/cref.

```
int i, j , k;
make_array<std::reference_wrapper<int>>(i, j, k) // int&
make_array(ref(i), ref(j), ref(k)) // int&
```

Le type retourné par make_array sera le type commun compatible avec les arguments passés, déterminé par std::common_type.

```
make_array(1, 2L); // long
make_array(1.0, 2.0f); // double
```

Utiliser shared_ptr et make_shared avec des tableaux

Les pointeurs intelligents permettent de prendre en charge la destruction automatique des objets dont ils ont la responsabilité. C'est un ajout majeur du C++11, puisqu'il permet de garantir l'exception-safe des pointeurs, contrairement aux pointeurs nus (c'est la même problématique que l'utilisation des tableaux style C présentée au-dessus) :

```
// incorrecte en C++
class A {
    int* p1;
    int* p2;
public:
    A() {
        p1 = new int;
        p2 = new int; // problème !
    }
    ~A() {
        delete p1;
        delete p2;
    }
};

// correcte en C++11, utilisez boost pour le C++03
class A {
    std::shared_ptr<int> p1;
    std::shared_ptr<int> p2;
};
```

La classe std::shared_ptr prend la responsabilité d'un objet partagé avec compteur de référence (l'objet est détruit uniquement si tous les pointeurs partagés sont détruits, ce qui garantie que l'on ne détruit pas un objet qui peut encore être utilisé). De plus, std::shared_ptr garantie l'accès multithread sur l'objet (concurrence). La fonction std::make_shared permet de créer un objet et de créer un pointeur partagé dessus. Il est également possible de créer un pointeur partagé sur un objet créé directement par new. Il faut dans ce cas faire attention que celui qui a créé l'objet (par exemple une bibliothèque style C) ne prenne pas en charge la destruction de l'objet, sous peine d'avoir une erreur pour double destruction. Il est également possible d'utiliser shared_ptr sur un tableau style C, avec quelques précautions : par défaut, shared_ptr appellera delete au lieu de delete[]. Il faut donc fournir un deleter particulier à shared_ptr (ou utiliser boost::shared_array) :

```
std::shared_ptr<int> p(new int[4], [](int* p){ delete[] p; });
```

Pour corriger ce problème (a priori, std::unique_ptr prend correctement en charge les tableaux), les drafts N3869 et N3870 proposent de modifier respectivement shared_ptr et make_shared.

```
auto p = make_shared<int[10]>(4); // crée 4 éléments
```

Les tableaux multidimensionnels

La manipulation de tableaux multidimensionnels est un problème classique en C++. Il est possible d'utiliser des solutions génériques, comme Boost.MultiArray, ou domaine spécifique, comme les outils de manipulation des images (Boost.GIL), la géométrie (Boost.Geometry) ou des matrices (Boost.uBLAS). Le draft N3851 propose d'ajouter ce type de tableau dans le C++. La création de tableaux à plusieurs dimensions pose un certain nombre de problématique sur :

- l'allocation des données en mémoire ;
- la gestion d'un nombre de dimensions quelconques (2D, 3D ou plus) ;
- la gestion des limites du tableau ;

- l'accès aux données du tableau, directement avec un index ou en utilisant les itérateurs (compatibilité avec les algorithmes de la bibliothèque standard).

Le draft actuel se base sur les deux éléments de conceptions suivants :

- l'allocation ne sera pas prise en compte pas les tableaux multidimensionnels, il faudra utiliser les conteneurs standard existant (vector, array, etc.) ;
- la manipulation des tableaux se fera via une vue, qui permet de manipuler un conteneur 1D comme un tableau à plusieurs dimensions.

Les différentes classes permettront de gérer les limites (std::bounds), les index (std::index), les vues (std::array_view et std::strided_array_view) et la linéarisation (std::bounds_iterator).

La gestion des limites (std::bounds)

La classe std::bounds permet représenter les limites d'un tableau sous forme carrée (c'est-à-dire que chaque ligne possède le même nombre d'éléments, chaque colonne possède également le même nombre d'éléments, etc. Ce que ne garantit pas par exemple vector<vector<T»). La définition de cette classe est la suivante (avec Rank le nombre de dimensions) :

```
template <int Rank> class bounds;
```

Pour créer un objet de ce type, plusieurs constructeurs sont possibles :

```
bounds<1> b1(5); // tableau 1D avec 5 éléments
bounds<3> b2({4, 5, 6}); // tableau 3D, avec 4 éléments dans
    // la première dimension, 5 éléments dans la seconde et
    // 6 éléments dans la dernière
```

Pour connaître le nombre d'éléments d'une dimension, utilisez l'opérateur [] :

```
b3[2]; // returne 5
```

Remarque : on voit ici un exemple de la puissance du mot-clé constexpr, ajouté dans le C++11. Les différentes implémentations actuelles de ce type de tableaux utilisent soit un accès à la compilation avec un template (comme par exemple avec get dans Boost.Geometry) :

```
geometry::get<geometry::min_corner, 0>(array);
```

soit un accès à l'exécution avec une fonction classique (comme par exemple avec [] dans Boost.MultiArray) :

```
bounds[2];
```

Donc deux syntaxes, selon les cas d'utilisation, ce qui complexifie le cas (particulièrement quand il est nécessaire de fournir les différentes combinaisons d'opérateurs). Dans le cas de std::bounds, l'opérateur [] existe en version constexpr :

```
constexpr const_reference operator[]
    (size_type component_idx) const noexcept;
```

Ce qui permet d'avoir une seule syntaxe aussi bien dans une expression évaluée à la compilation qu'à l'exécution. Une seule syntaxe utilisable dans les différentes configurations au lieu d'une syntaxe spécifique selon les cas.

```
foo<b1[1]>(); // compile time
foo(b1[1]); // runtime
```

Conclusion : pensez à utiliser constexpr.

Pour revenir à std::bounds, cette classe propose différents opérateurs et fonctions classiques (le Rank doit être identique) :

- opérations de comparaison : b1 == b2, b1 != b2 ;
- opérations arithmétiques avec un scalaire : b * 2, b / 2, b *= 2, b /= 2, 2 * b (remarquez bien que seul l'opérateur * est commutatif) ;

- opérations arithmétiques avec un index (voir la suite pour std::index) : b + i, b - i, b += i, b -= i, i + b (remarquez également que seul l'opérateur + est commutatif) ;
- pour connaitre les dimensions : size(), contains(index) ;
- pour les itérateurs : begin(), end().

La gestion des index (std::index)

La classe std::index permet de gérer un élément dans un tableau. Sémantiquement, cela correspond à un vecteur dans un espace à plusieurs dimensions par rapport à l'origine. Cette classe est proche de std::bounds et propose un certain nombre de fonctionnalités similaires :

- opérations de comparaison, arithmétiques avec un scalaire, connaître les dimensions et les itérateurs ;
- en complément, std::index ajoute les opérateurs d'incrémentation et décrémentation : +i, -i, ++i, –i.

La gestion des vues

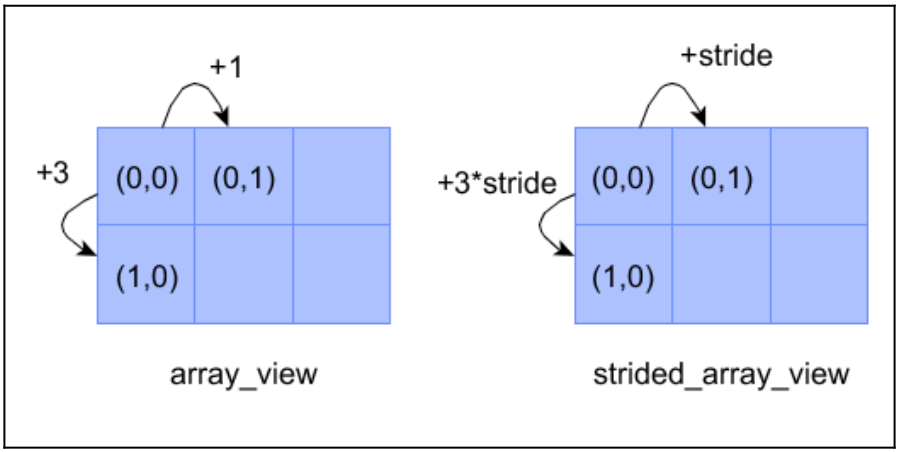
Pour la gestion des vues, vous avez la possibilité d'utiliser deux classes : std::array_view et std::strided_array_view.

```
template <typename ValueType, int Rank = 1>
class array_view;

template <typename ValueType, int Rank = 1>
class strided_array_view;
```

La première permet de gérer des conteneurs dont les données sont contiguës, avec une distance de 1 entre les éléments de la dernière dimensions. Par exemple, pour un tableau 2D de dimensions [2, 3], incrémenter de 1 permet de passer à l'élément sur la colonne à droite, incrémenter de 3 permet de passer à la ligne en dessous. La seconde

permet de gérer des conteneurs dont les données sont placées avec une distance de stride pour les éléments de la dernière dimension. Par exemple, pour le même tableau 2D que précédemment, il faut incrémenter de stride pour passer à l'élément sur la colonne de droite et de 3*stride pour passer à la ligne en dessous.



Ces deux classes proposent une interface similaire. La création se fait en fournissant un conteneur (en général std::vector) et les dimensions (optionnel si une seule dimension).

```
auto v1 = vector<int>(5);
auto view1 = array_view<int, 1>{v1}; // 1D
auto view2 = array_view<int, 2>{{2, 5}, v1}; // 2D
auto view3 = array_view<int, 1>{view1}; // copie

char v2[3][1][2]; // tableau 3D style C
auto view4 = array_view<char, 3>{v2}; // 3D

auto v3 = array<array<float, 4>, 5>; // tableau statique 2D
auto view5 = array_view<float, 2>{v3}; // 2D

auto v4 = vector<float>{4, vector<float>(5)}; // tableau
    // 2D non contiguës en mémoire
auto view6 = strided_array_view<float, 2>{v4}; // strided 2D
```

La copie permet de créer une seconde vue sur un conteneur, les données du conteneur ne sont pas copiées.

L'accès aux données se fait en utilisant l'opérateur [], qui prend un index en paramètre. Il n'est pas obligatoire d'appeler explicitement le constructeur de std::index, celui-ci est appelé implicitement.

```
view1[2] = 4; // écriture
int i = view1[3]; // lecture

view2[{1, 2}] = 5; // accès 2D
```

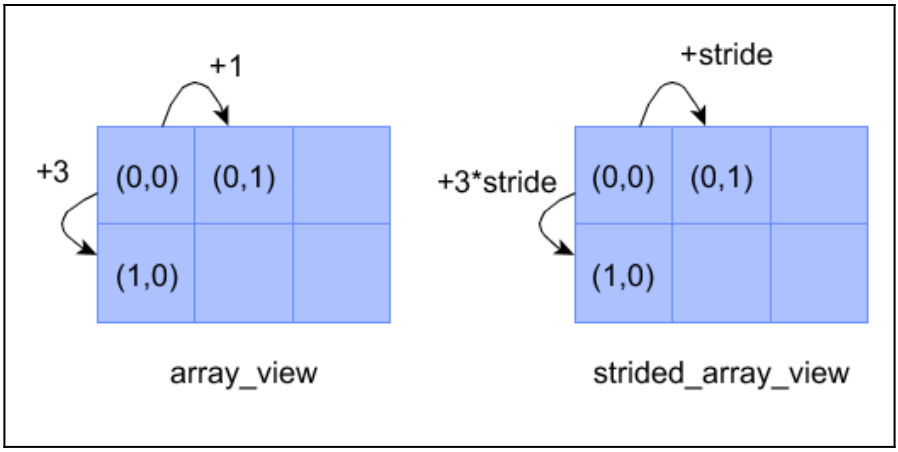
Ces deux classes proposent plusieurs fonctions d'accès aux limites, taille, stride et données :

```
constexpr bounds_type bounds() const noexcept;
constexpr size_type size() const noexcept;
constexpr index_type stride() const noexcept;
constexpr pointer data() const noexcept // pour array_view
```

Le slicing

Le slicing est une opération permettant d'obtenir un tableau de dimension (N-1) à partir d'un tableau de dimension N (voir l'image suivante), en fixant la première dimension (un tableau (x,y,z) donnera un tableau (y,z) par slicing). Cette opération est réalisée en utilisant l'opérateur [] avec un index de dimension 1.

```
auto view4D = array_view<int, 4>{{4, 4, 4, 4}, v}; // 4D
auto view3D = view4D[2]; // 3D
```



Le slicing permet en particulier d'utiliser une syntaxe similaire aux tableaux multidimensionnels actuels, les deux lignes suivantes sont identiques :

```
view[1][2][3] = 42;
view[{1, 2, 3}] = 42;
```

Le sectionning

Cette opération permet d'extraire un sous tableau de dimension N à partir d'un tableau de dimension N (voir image précédente), utilisant la fonction section et en précisant l'origine et les dimensions du nouveau tableau. Le nouveau tableau est de type strided_array_view, excepté si le tableau d'origine est un array_view et si le nouveau tableau ne redéfinit pas de nouvelles limites.

```
auto strided_section = view.section({1, 2}, {2, 3});
auto section = view.section({1, 2});
```

Linéarisation avec bounds_iterator

Pour terminer, il est parfois nécessaire de linéariser un tableau, pour que l'ensemble de ses éléments soient vu comme une seule collection 1D (par exemple avec les algorithmes standards). La différence avec un itérateur obtenu avec v.begin() et v.end() est que les bounds_iterator respectent les limites et strides des tableaux.

Linéarisation avec bounds_iterator Linéarisation avec bounds_iterator Cet itérateur est un BidirectionalIterator constant, qui peut être déréférencé pour récupérer un index sur la vue. Cet index peut être alors utilisé pour lire ou modifier le tableau.

```
auto first = begin(view.bounds());
auto last = end(view.bounds());

// avec for
for (auto it = first; it != last; ++it) {
    view[*it] *= 2;
}

// avec std::transform
transform(first, last, first, [&](auto idx){ view[idx] *= 2;
});

// avec range-based for
for(auto idx : view.bounds()) {
    view[idx] *= 2;
}
```

Conclusion

Remarque importante : les fonctions présentées dans cet articles ne sont pas encore implémentées dans GCC, je n'ai pas testé les codes d'exemple donnés dans cet article. Certaines images et codes d'exemples proviennent directement des drafts du comité C++.

Mise à jour

- Mise à jour du draft N3976 “Multidimensional bounds, index and array_view, revision 2”, qui remplace N3851.
  - Ajout d'un header <array_view>, qui contient les classes `array_view` et `stride_array_view` ;
  - Ajout d'un header <coordinate>, qui contient les classes utilitaires permettant de gérer des coordonnées (index, bounds et bounds_iterator).
- Correction : suppression syntaxe du constructeur avec déduction de l'argument template (où j'ai pondu un truc pareil…)

C++, C++1y