hmidoz

Synthèse de CPP

Hello World ! (Chaptre 1 et 2)

- L'affichage en CPP se fait en injectant dans des flux de fichier ou standard, le flux d'affichage standard est le flux **cout.**
- La fonction **main** est une fonction qui renvoie un **int**, on peut faire un **return**, ou pas.
- En Java, on importait des classes ou des API, en C++ on utilise une directive de **préprocesseur** pour inclure des classes ou des API via la directive **#include** suivie d'une API entre <> ou un fichier .h entre '' ''.
- En C++, on utilise des espaces de noms, qu'on définit via le mot clé **namespace** et qui permettent d'englober des fonctions/méthodes dans un espace de nom. Lors de leur utilisation, nous devront déréférencer le/la fonction/méthode/attribut en faisant précéder par **espacedeNom::.**
- Pour pouvoir rendre le code plus léger, on peut mettre en début de code qu'on utilise un namespace, ainsi si le compilateur ne connaît pas la fonction, la méthode ou l'attribut, il fera précéder par le déréférencement du namespace. Pour dire qu'on utilise un namespace, on met **using namespace nomNamespace;**
- Si on ne met pas ces namespace et qu'on ne déréférence pas, il y'aura des **erreurs de compilation.**
- Notez qu'on peut également dire qu'on utilisera un namespace pour une méthode/fonction/attribut en particulier et juste pour celle-ci, on le fait en mettant en début de programme **using nomNamespace::fonction ;**

Les types (Chapitre 3)

- Le C++ est un langage très **typé**, ce qui signifie qu'avant d'utiliser une variable, on devra préciser son type.
- Voici les différents types simples en C++ : **int, double, char, short, long, long long, bool, float, long double.** A noter qu'on peut préciser que le type est **signed** ou **unsigned**.

- Chaque type a une taille qui lui est attribuée en mémoire, qui peut être variable en fonction du type. Par exemple, on sait que la taille d'un **int** vaut **4 bytes.**
- Pour savoir la taille d'un type ou d'une instance en mémoire, on utilise l'opérateur **sizeof(nomVariable)** ou **sizeof(nomType).**

- **Petite parenthèse sur les modificateur de flux:** Il existe des fonctions qui permettent de modifier l'affichage dans le flux, comme par exemple **boolalpha** qui transformera les booléens, affichés normalement en 0 pour faux ou 1 pour vrai, qui seront affichés dorénavant en **true** ou **false. Tout le reste du flux sera altéré**, sauf si on applique la fonction **noboolalpha** qui permet d'annuler ces effets. Il existe plein de modificateur de flux, nous ne les utiliserons pas souvent.

- Parlons maintenant d'une chose très importante, les **conversions**. En C++, toutes les conversions sont possibles à condition de ne pas utiliser les accolades. Ainsi si on veut transformer un **double** en **int**, on enlevera la partie à virgule, si on veut transformer un **char** en **int** et le mettre dans un **char**, c'est possible.

- Lors de l'utilisation d'accolades, le compilateur détectera les **conversions dégradantes.** Ainsi, si on veut initialiser un int avec un double, il y'aura une erreur de compilateur, car on perds la partie a virgule. Si on veut transformer un unsigned en int, on perd une grande partie de valeurs car tout est en non-signé. Par contre, les types short et int étant très proches, ils n'y aura pas de conversion dégradante, car c'est une **promotion** quand on passe de short a int. Le contraire donnerait également une conversion dégradante.

- Lors de l'utilisation de parenthèses ou de =, il n'y aura aucun problème car le compilateur ne vérifie pas si la conversion est dégradante.

- Lors d'une initialisation, si on initialise avec des accolades vides, la valeur par défaut sera 0. Si on initialise avec des accolades vides c'est un **prototype de fonction**. Il faut donc être très attentif car « **int y()** » ne posera pas de problème au compilateur, mais si on essaye de lui donner une valeur par la suite, erreur de compilation. L'affichage par contre, nous montrera 1, la valeur de **true**, car on fait rien dans la fonction. Pour finir, si on n'initialise pas la variable et qu'on crée simplement, la valeur sera bizarre et arbitraire.

Signed et Unsigned (chapitre 4)

- La première chose à savoir c'est que les types signed et unsigned ne sont pas applicable pour tous les types. Ainsi **bool, float, double** et **long double** n'ont pas ces spécifications et si on essaye de les mettre signed ou unsigned devant un de ces types, erreur de compilation.
- Certains types comme **char** ne sont ni signed, ni unsigned si on les a déclaré uniquement comme char. Si par contre on les déclare comme **signed char** ou **unsigned char** alors ils seront signed char ou unsigned char mais plus **char** tout court. Donc quand on met char, on ne sous-entend ni unsigned, ni signed.
- D'autres types comme **int** le sont pas défaut, ainsi un int normal sera comme un signed int, par contre un unsigned int ne sera pas considéré comme un int normal. Donc quand on met **int**, on sous-entend **signed int**, de même quand on met juste **unsigned** comme type, on sous-entend **unsigned int**.

Les types immédiats (chapitre 5)

- La question qu'on se pose, c'est comment savoir quel type correspond à quel immédiat. Quand on met le chiffre 4, ça pourrait être un short, un long, un int… Voyons comment le compilateur comprend tout cela.
- Un nombre entre ' ' comme '4' par exemple, sera toujours interprété comme un **char** par le compilateur.
- Un nombre simple comme 4, sous forme décimale ou dans une autre base sera toujours interprété comme un **int** (donc un signed int).

- Un nombre dans une base décimale ou une autre base suivie d'une **U**, par exemple 4U sera toujours interprété comme un **unsigned int.**
- Ainsi un nombre suivi de **L** sera interprété comme un **long**, un nombre suivi de **LL** comme un **long long** et si on fait précéder le L ou le LL par un U, on parle d'**unsigned long** ou **unsigned long long**.
- Un nombre suivi d'un «.» comme « 4. » ou d'un e suivi d'un chiffre comme par exemple **4e0** sera considéré comme un double. Si on fait suivre les mêmes écritures par un F comme « 4e0F » alors le type sera **float**, si c'est par un L comme dans « 4.L » alors c'est un **long double**.

Le type booléen en numérique (chapitre 6)

- Lors de sa création, un **booléen** vaut 0, même si il n'a pas été **initialisé**. Et 0 vaut false comme on le sait déjà.
- Lorsqu'on donne une valeur différente de 0 au booléen, il devient true et sa valeur passera à 1 QUOIQU'ON FASSE. Même lors de l'affichage ça vaudra 1.
- **Un nombre est une expression booléenne** car le type bool est un type numérique. Donc 15 est une expression booléenne au même titre que 1, true, -786865875 ou .0.

Le static cast (chapitre 7)

- La première chose qu'on doit savoir, c'est que lorsqu'on divise un entier par un entier, ça fera une **division entière**, c'est logique. Par contre, lorsque l'un des deux opérandes est de type **double** ou **float**, ça deviens une **division décimale** via ce qu'on appelle un **transtypage implicite,** c'est-à-dire qu'on passe d'un type a un autre sans le demander formellement. Mais pour le demander formellement, on procède a l'usage de **cast**, des conversion. Et un de ces cast est le **static cast**.
- Ce static cast nous permet par exemple de faire une demande implicite, pour cela on l'écris comme « **static_cast<type>(variable)** »
- Ainsi on pourra demander de changer un type en un autre pour l'affichage, le calcul ou la conversion toute simple
- Pour finir, une parenthèse sur le modificateur de flux **showpoint** qui permet de voir tout ce qu'il y'a avant ou après la virgule lors de l'affichage.

Les fonctions (chapitre 8, 9, 10 et 11)

- Pour commencer une fonction est une partie de code qu'on appellera dans le main pour l'exécuter. En C++, chaque fonction, sauf le main, doit avoir ce qu'on appelle un **prototype**, c'est-à-dire l'entête de la fonction à la différence près qu'on peut mettre seulement le type des paramètres sans indiquer leur nom de variable. Ils permettront au compilateur de dire quelle fonction il y'a dans le code.
- Ces prototypes sont séparés du code. Une bonne pratique en C++ est de les exiler dans un fichier de **header** (.h) qui a le même nom que le fichier source. Ensuite on l'inclura via la directive **#include** en mettant son nom entre ''.
- L'appel de ces fonction se fait dans le main en mettant leur nom suivi de () qui comprends ses paramètres si il y'en a. Si on ne met pas suffisamment de paramètres, erreur de compilation, sauf si on a définis dans le prototype un **paramètre par défaut**.
- Si on ne choisis pas de mettre les entête, c'est une mauvaise pratique, mais si la fonction n'est pas définie avant le main alors il y'aura une erreur de compilation. Par contre l'utilisation de prototype permet de les définir ou on veut dans le fichier source.
- Si on fait une erreur dans le prototype, en indiquant par exemple le mauvais type de retour, il y'aura une erreur de compilation car la fonction **int f()** n'est pas la fonction **void f().**
- Si on indique un type de retour sans rien renvoyer, la réponse sera étonnante, en effet on aura un chiffre totalement bizarre qui viendra prendre la place de ce qui n'a pas été renvoyé.
- Le dernier cas vu ici est le cas où l'on indique le prototype de la fonction mais qu'on ne la définis pas, bien entendu il y'aura une erreur de compilation, le compilateur cherchant la fonction mais ne la trouvant pas.
- La bonne pratique c'est de séparer les fonctions dans un fichier .cpp, de donner le même nom au fichier qui contient le .h et d'inclure le fichier d'entête dans le .cpp et le main de façon, ainsi tout fonctionnera sans soucis.

Les paramètres et valeurs par défaut (chapitre 12 et 13)

- On peut également définir des fonctions ayant des paramètres qui ont des **valeurs par défaut**. Cela se fait dans le fichier **header**, on met dans le prototype un « = valeur » juste après le type du paramètre. Voici un exemple : **void f(int = 45).** Ici lors de l'appel de f, si on ne lui donne pas de paramètre, le paramètre de l'int vaudra 45. On pourrait malgré tout définir le paramètre en lui donnant une valeur lors de l'appel.
- Par contre, si on a plusieurs paramètres par défaut, un problème peut se poser. Si on choisit de donner un paramètre sur deux, lequel ira auquel ? Une chose à savoir c'est que les paramètres par défaut sont **évalués** de **droite à gauche**, celui qui est à droite est prioritaire, et ainsi de suite. Donc si je donne 3 paramètres à ma fonction, le premier ira a celui le plus à droite, le deuxième a celui juste à sa gauche et le dernier juste a la gauche du précédent.
- Etant donné le fait que les paramètres sont évalués de droite à gauche, on doit mettre **tous les paramètres par défaut les uns à la suite des autres en partant de la droite**, si il y'a un « trou » avant un autre paramètre par défaut, erreur de compilation.
- De même, si on définit les paramètres par défaut dans le fichier .cpp et donc pas dans le header, pas d'erreur de compilation, mais ils ne sont pas pris en compte.

Les fonctions récursives (chapitre 14)

- Pour finir parlons d'un concept nommé les **fonctions récursives**. Pour faire simple, une fonction récursive est une fonction qui fait appel à elle-même et forme une sorte de boucle car elle s'appelle elle-même, qui s'appelle encore elle-même et ainsi de suite. Si cette fonction ne possède pas de condition pour qu'elle termine, il y'aura une erreur à l'exécution.
- Si cette fonction possède une condition pour qu'elle termine, les effets de la fonction seront additionner et chaque return prendra en compte tous les return qui la précède, voici deux exemples aux point suivant.

- ```
  unsigned accumulate(unsigned nb)
  {
      if (nb == 0)
      {
          return 0;
      }
      else
      {
          return nb + accumulate(nb - 1);
      }
  }
  ```

  Avec nb = 10 celà donne :

  1er appel : return 10 + accumulate 9 6eme appel : return 5 + accumulate 4
  2eme appel : return 9 + accumulate 8 7eme appel : return 4 + accumulate 3
  3eme appel : return 8 + accumulate 7 8eme appel : return 3 + accumulate 2
  4eme appel : return 7 + accumulate 6 9eme appel : return 2 + accumulate 1
  5eme appel : return 6 + accumulate 5 10eme appel : return 1 + 0

  10+9+8+7+6+5+4+3+2+1 = 55

- ```
  unsigned factorial(unsigned nb)
  {
      return ((nb == 0) || (nb == 1)) ? 1 : nb * factorial(nb - 1);
  }
  ```

  avec nb = 4 ça donnera :

  1er appel : 4*factoriel(3)
  2eme appel : 3*factoriel(2)
  3eme appel : 2*1

  1*2*3*4 = 24

Les fonctions surchargées (chapitre 15, 16, 17,18 et 19)

- Comme dis précédemment, deux fonctions ne sont pas les mêmes si leur type de retour n'est pas pareil, il en va de même si les paramètres ne sont pas les mêmes. En effet, lorsque deux fonctions ont le même nom mais pas les mêmes paramètres, on dit qu'on fait une **surcharge de fonction**.
- Mais alors, comment le compilateur sait à quelle fonction on veut faire appel ? Tout simplement avec le **type des paramètres**. En effet, en fonction des types **le compilateur fera son choix**, mais dans la pratique cela posera un souci, en effet, on peut passer de n'importe quel type a un autre, donc ce n'est pas parce que ma fonction demande un char que je ne peux pas lui donner un booléen.
- Donc si on a une seule fonction, le choix sera automatiquement fait vers cette fonction, qu'on envoie un **long**, un **int**, un **bool**, un **char**, on fera **toujours appel à cette fonction**, c'est lorsqu'il y'en a plusieurs que les choses se gâtent.
- En effet, avec deux prototypes, le compilateur doit faire un choix, mais parfois il ne peut pas faire le choix car aucune proposition ne se démarque. Le compilateur a des priorités, elles sont détaillées ici.

- 1) La **correspondance exacte**, c'est ç dire quand le type qu'on envoie est le même que le type de la fonction, le choix sera fait pour cette fonction.
- 2) **template** (pas détaillé ici)
- 3) la **promotion numérique**, quand nous avons un des types suivant : **char, signed char, unsigned char, short, unsigned short** et **bool** avec une fonction **int** ou **unsigned**, cette fonction aura toujours la priorité, même chose pour de **float** a **double**, la fonction double aura la priorité.
- 4) les **conversions standard**, quand on passe d'un type a l'autre
- 5) **conversions définies** par l'utilisateur.
- Si à un moment donné, le compilateur a le choix entre deux fonctions et que malgré ces règles **il n'arrive pas à départager**, erreur de compilation. Cela arrive souvent lorsque le compilateur arrive au 4) avec les conversions standard, la conversion est possible pour toutes les fonctions du coup, problème.
- Voici un exemple complet :

  void f(char) { cout << ''f(char)'' << endl}
  void f(long double) {cout << ''f(long double)'' << endl}
  void f(double) {cout << ''f(double)'' << endl}
  void f(short) {cout << ''f(short)'' << endl}

  f(4) → erreur de compilation, choix entre f(char) et f(ong double)
  f(4.0L) → affichera f(long double), **correspondance exacte**
  f(4.0F) → affichera f(double), **promotion numérique** float -> double
  f(true) → erreur de compilation, **ambiguïté** entre les 4 fonctions

Directive de préprocesseur (chapitre 20)

- Comme on le sait déjà, **#define** est une directive de préprocesseur, tout comme **#include** et toute les fonctions précédée par ce #. Ce sont des commandes qui sont effectuées avant la compilation et qui modifient le code. #include permet d'incorporer des fichiers en début de programme et #define permet de définir un **macro**. On les écrit généralement en majuscule.

- Ce macro sera définis avec une valeur qui le remplacera partout où on le mettra dans le code. Par exemple, **#define BROL -5** à chaque fois qu'on mettra BROL dans le programme, il sera remplacé par -5. Ce n'est pas utile dans ce cas, mais ça pourrait l'être dans d'autres.
- Il existe bien entendue d'autre directive, **#if, #ifndef, #ifdef, #endif** etc… Qui ont chacun un rôle précis.

Les constantes (chapitre 21)

- On sait qu'une variable **constante** est une variable qui, une fois initialisée, ne changera pas de valeur. Pour déclarer quelque chose de constant, on utilise le mot clé **const**. En C++, une constante doit obligatoirement être initialisée. Si elle n'est pas initialisée, erreur de compilation. Pareil, si on essaye de modifier la valeur en assignant ou en incrémentant par exemple, erreur de compilation.
- On a également la possibilité d'initialiser des variables non constantes via une expression mathématique comme un calcul à partir d'une **variable constante**, cela ne posera aucun souci, sauf si on déborde la variable (en donnant par exemple + que 128 à un char, car il faudra faire un calcul pour connaître la vraie valeur. Cela posera également problème si il y'a une variable non constante dans l'expression bien entendu, dans ces deux cas, il y'aura erreur de compilation.
- Si on essaye d'initialiser une **variable** à partir **d'une variable non constante**, c'est-à-dire une **variable inconnue à la compilation** (connue à l'exécution), erreur de compilation. Pareil pour les variables constantes qui sont inconnus à la compilation (par exemple une variable constante initialisée avec une fonction random).
- On peut initialiser une constante avec une variable, aucun souci avec ça, c'est le contraire (une variable avec une constante) qui pose problème.
- Lors de la création de la variable, qu'on écrive **const int** ou **int const** ne change rien, le type est le même (int). Ainsi si on utilise **typeid** avec int et const int, il nous dira que c'est le même type, par contre, avec **is_same** (classe qui compare les types) , il nous dira que ce ne sont pas les mêmes types (car il prend en compte le mot clé const, contrairement à typeid qui s'en fous), par contre, tout le monde s'accorde à dire que const int et int const sont les mêmes types.

Les variables constexpr (chapitre 22)

- Le mot clé **constexpr** est un mot clé qui détermine également quelque chose de constant, mais ce n'est pas la même chose que **const.** Au niveau des variables, la différence se situe dans le fait que les variables **constexpr**, contrairement aux variables const doivent toujours être initialisé avec une expression **constante**, dans une variable **const** ou non.
- On ne pourra pas initialiser tout le temps avec une variable **const** du coup, car les variables **const** peuvent contenir eux même des expressions ou variables non constantes.
- Comme pour const, les variables doivent être initialisé et ne peuvent pas être modifiée, sinon erreur de compilation.
- Une variable **constexpr** peut bien entendu être initialisée à partir d'autres variable constexpr.
- Contrairement à const, ou l'on pouvait initialiser avec des fonctions, une variable constexpr ne peut être initialisée qu'avec des **fonctions constexpr**, sinon erreur de compilation, même si on essaye de passer par une variable const tiers.
- On ne peut pas utiliser is_same ou typeid sur le type constexpr.
- Le but de l'utilisation de constexpr, c'est que ça utilise moins de mémoire qu'une variable et que le programme aura des meilleures performances.

Les fonctions constexpr (chapitre 23, 24 et 25)

- Ce mot clé peut également être utilisé pour les fonctions. En C++11, les **fonctions constexpr** sont constituée d'un seul return, sinon erreur de compilation.
- On pourra appeler ces fonctions pour faire généralement des calculs, comme transformer des degrés Celsius en degré Kelvin. On pourra mettre leur résultat dans des variables classique ou des variables **constexpr**, qui peuvent être initialisés avec.

- Au niveau des **performances**, elles peuvent être intéressantes, surtout quand on stocke leur valeur finale dans des variable **constexpr** ou **const** alors **le calcul ne prendra une seule nanoseconde pour se faire** à l'exécution, alors qu'avec des variables classiques le compilateur aurait dû faire des calculs à l'exécution. Ce qui nous permet de le vérifier, ce sont les méthodes de chrono ainsi qu'une fonction récursive.

Les références de variables (chapitre 26)

- Une **référence** est une **sorte de lien** avec une variable. En effet une variable peut être **pointée** par une **référence** et la référence sera une sorte d'**alias** de la variable, elle occupe le **même espace mémoire** que la variable à laquelle elle est liée. Les références ont également un **type**, tout comme les variable et on les reconnaît grâce au **&** qui précède leur nom.
- La référence doit être du même type que la variable à laquelle elle est liée, sinon erreur de compilation.
- Une référence doit toujours être initialisée, sinon, erreur de compilation.
- Si on cherche à afficher une référence, cela affichera exactement la même chose que la variable liée.
- Elles occupent le même **espace mémoire**, donc si on cherche à savoir leur adresse en faisant précéder le nom de variable ou de référence par **&**, on aura les mêmes valeurs.
- Qu'on modifie la référence ou la variable liée, les deux seront modifiés.
- Une référence n'est pas obligatoirement liée à la même variable pour toujours, on peut la liée à une autre variable si elle est du même type, sinon, erreur de compilation.
- On ne peut pas lier une référence à un **immédiat** ou une **expression**. On peut uniquement le faire avec une variable.
- La **taille** de la variable sera égale à la taille de la référence.
- Lorsqu'on utilise l'opérateur **typeid** pour comparer les types, on se rend compte que typeid dis que la référence est du même type que le type classique de la variable, par exemple que int = int&. Par contre **is_same** lui vois bien la différence entre les deux types int et int&.

Référence vers une variable constante (chapitre 27)

- Vu que le type de la variable est désormais **const** une référence sans const ne suffit plus, si on tente de le faire, erreur de compilation. On appelle cela une **référence constante.**
- Toujours aucune différence entre le fait que le type soit **const type** &ou **type const &**.
- Vu que la variable est constante, qu'on tente de la modifier elle ou sa référence c'est la même chose et ce n'est pas autorisé, on risque l'erreur de compilation.
- Une référence constante peut également être liée à une variable **constexpr**, étant donné que les références **constexpr** n'existent pas, erreur de compilation si on le test.
- Le fait qu'une **référence constante** est constante ne l'empêche pas d'être liée à une variable non constante. Si on modifie la variable il n'y aura pas de soucis, par contre la **modification de la référence** entraînera une erreur de compilation.
- Alors qu'une référence vers un immédiat était **interdite**, une référence constante vers un immédiat ne posera **aucun problème**. D'ailleurs, si deux références constantes pointent vers le même immédiat, alors **l'adresse des références sera la même**.
- Petite parenthèse sur les **fonctions constexpr**, leur résultat peut être stocké dans une **variable constexpr** seulement si les paramètres ne sont pas des variables. Par contre avec des **références const** (qui acceptent les constexpr), le stockage du résultat de la fonction ne posera aucun problème avec des variables en tant que paramètre ou pas. A noter qu'une **référence non const** aurait **refusé** une fonction constexpr quoiqu'il arrive.
- Parlons maintenant de l'opérateur **const_cast.** Nous avions déjà vu le **static_cast** qui transformait une variable d'un type à un autre. L'opérateur const_cast permet de transformer une variable d'un type à un autre en sautant la limitation du **const**. Si on essaye de faire un static_cast sur une variable const, erreur de compilation. Mais c'est extrêmement dangereux, car le résultat est imprévisible. Ça peut bien se passer comme très mal se passer.

- En sautant cette **limitation**, on pourra **mettre une variable constante dans une référence non constante**. Dès lors on pourra essayer de **modifier la référence et ainsi modifier la constante**… Mais un résultat bizarre se produit ! La référence continue d'être liée à la constante et s'est modifiée, mais la **constante ne s'est pas modifiée**, ce qui va à l'encontre de tous les principes vu depuis le début des références, c'est pour ça que c'est dangereux.
- Par contre, pour faire ça proprement, on peut utiliser un const_cast au moment de modifier la référence, on saute les règles de la constante, on modifie la référence et la constante et le résultat est prévisible. Exemple : en faisant ++ const_cast<int &>(var).

Référence et fonction (chapitre 28)

- On peut également utiliser des **références en paramètre** dans une **fonction** ou même en **type de retour** sans qu'il y'ait de problèmes. Si on en utilise en paramètre en envoyant des variables normales il y'aura aucun soucis et l'effet est comme celui de faire **plusieurs return** car c'est la **référence qui est modifiée** (donc l'adresse en mémoire) et pas une copie qu'on renvoie par la suite.
- On pourrait également faire une surcharge de fonction, une avec le type **int &** et **const int &** comme dans l'exemple du cours, le choix alors sera à chaque fois fait en fonction que ça soit une constante ou pas. En envoyant une **variable normale** ou une **variable pré-incrémentée** c'est **int &** qui est appelé alors que c'est **const int &** qui est appelé si on envoie **une constante**, **un immédiat**, **une variable crée juste pour l'appel**, **une expression**, **une post-incrémentation** ou même une **variable d'un autre type** sous toutes les formes précédentes ! L'utilisation de **const int &** permet de meilleures performances.
- Si la fonction retourne une **référence**, c'est comme si elle retournait une **variable réelle** et non pas une **valeur**. Du coup on peut même utiliser l'appel de fonction comme une variable en faisant par exemple f(i) = 2;

Référence et surcharge de fonctions (chapitre 29, 30, 31, 32 et 33)

- Une fonction ayant besoin d'une **référence non constante** en paramètre ne pourra pas recevoir d'**immédiat**, d'**expression**, de **constante** ou de **variable/immédiat** d'un autre type, sinon erreur de compilation.

```
void f(int & i)
{
    std::cout << "f(int &) (i : " << i << ")";
}
```

- Une fonction ayant besoin d'une **référence constante** en paramètre pourra recevoir des **immédiats**, des **expressions**, des **constantes**, des **variable d'autres types** sans problèmes. Par contre lorsqu'on fait une **surcharge** avec une fonction ayant besoin d'une **référence non constante**, seuls les variables non constants du type de la référence choisiront cette fonction, car le principe de la **correspondance exacte** est appliqué ç (exemple avec un entier i : f(i) → ref non const

```
void f(int & i)
{
    std::cout << "f(int &) (i : " << i << ")";
}

void f(const char & c)
{
    std::cout << "f(const char &) (c : " << static_cast<int>(c) << ")";
}
```

- Lorsqu'on **surcharge** une fonction ayant besoin d'une **variable entière** par exemple avec une fonction ayant besoin d'une **référence d'entier**, cela n'a **pas beaucoup de sens**. En effet, à part pour l'appel avec la variable entière qui posera un **problème d'ambiguité** et donc une erreur de compilation, toute les autres fonctions **choisiront la fonction avec la variable entière et jamais celle avec la référence**.

```
void f(int i)
{
    std::cout << "f(int) (i : " << i << ")";
}

void f(int & i)
{
    std::cout << "f(int &) (i : " << i << ")";
}
```

- SI on **surcharge** une fonction ayant besoin d'un **entier** avec une fonction ayant besoin d'une **référence d'entier constant**, **tous les appels seront ambigus**, que ça soit avec un immédiat, une variable, un autre type…

```
void f(int i)
{
    std::cout << "f(int) (i : " << i << ")";
}

void f(const int & i)
{
    std::cout << "f(int &) (i : " << i << ")";
}
```

- Pour finir, si on utilise une **fonction qui reçois un entier** et **renvoie une référence**, c'est **idiot** car ça ne sers a rien. En effet, si on veut retourner le paramètre alors **on retournera la référence de la copie**, du coup **l'original ne sera pas modifié**, donc à moins qu'on stocke dans une autre variable, ça ne sers a rien. Par contre si on change le paramètre en référence alors tout s'éclaire et c'est magique.

```
int & f(int i)  // changer (int i) en (int & i)
{
    i /= 2;
    return i;
}
```

Les pointeurs sur variables (chapitre 34)

- Les **pointeurs** sont un concept primordial en C++. Un pointeur est représenté par l'utilisation de *****, à la déclaration. Un pointeur est un lien qui ramène vers une variable, un objet voir une fonction. Les pointeurs, tout comme les **références**, ont un type et doivent **pointer** une variable de même type. Un pointeur d'**int** ne pourra pas pointer un double.
- L'**initialisation** ne pose aucun problème de compilation, **on peut ne pas initialiser un pointeur**, l'initialiser avec des accolades vides mais le plus correct est de l'initialiser avec **nullptr**.
- Le **pointeur** doit contenir l'**adresse** vers laquelle il doit pointer en mémoire, du coup, on ne lui donne pas une variable de type int toute simple par exemple, mais une **référence** avec l'utilisation de **&**. On pourra également lui attribuer un autre pointeur.
- Si on **affiche un pointeur**, c'est l'adresse qui sera affichée. Pour voir ce que pointe la variable on doit **déréférencer le pointeur** en utilisant ***** avant de citer le pointeur comme ça : ***pointeur.**
- On pourra également utiliser le référencement dans une variable pour le stocker dans une variable.
- Il ne faut pas confondre l'opérateur * pour les pointeurs avec celui de la multiplication, ainsi *pointeur * -*pointeur serra la multiplication entre deux valeurs pointée.
- On ne peut pas déréférencer un pointeur s'il a été assigné avec **nullptr**, sinon erreur de compilation.
- De même, on ne pourra pas donner une valeur entière ou comme ça a un pointeur en faisant **pointeur = 12**, on touche a de la mémoire qui n'appartiens pas au programme et ça peut mal finir. Donc erreur de compilation. Un **static_cast** en *int ne marchera pas non plus et donnera également une erreur de compilation. La seule manière qui marche est un nouveau **reinterpret_cast<*>** marchera mais le résultat sera bizarre et il est fort probable d'avoir une erreur a l'exécution lors du déréférencement.
- Parlons des pointeurs **génériques**, ce sont des pointeurs de type void * qui feront un transtypage implicite lorsqu'on les fera pointer sur une variable. Si il y'a déréférencement, on aura une erreur de compilation

- L'utilisation des pointeurs générique est donc couplée à un ***static_cast<type*>** pour déréférencer la variable tout en la transformant. On pourra même assigner avec cette forme de pointage sur un **static_cast**. A noter que si on modifie cette valeur, on modifie bien entendu l'original et c'est normal.
- Si on essaye d'assigner à un déréférencement de pointeur générique, erreur de compilation.
- On pourra faire **pointer un autre type** après avoir fait pointer un premier type. Par exemple, on peut pointer un int, puis un double.
- Parlons maintenant des constantes et des pointeurs constants. En effet, on peut **pointer des variables const** mais pas avec n'importe quel pointeur**. Il existe des pointeurs constants,** ces pointeurs ne pourront pas être assigné à une autre variable et du coup, si on essaye de le modifier en assignant à une autre variable, erreur de compilation. également être obligatoirement initialisé, si on ne le fait pas, erreur de compilation. **Par contre, rien n'empêche de modifier le déréférencement**.
- On déclare ces pointeurs constants comme **type * const pointeur.** Il existe d'autres pointeurs qui sont des type const * ; des constants pointeurs. On les écrit sur sous forme **type const *** et si on essaye de modifier la valeur qu'ils pointent en faisant un déréférencement, il y'aura une erreur de compilation. Des pointeurs constants peuvent également pointer d'autres pointeurs constants et pointer vers la même valeur. **Ces pointeurs permettent de pointer des variables constantes.**
- Il existe également des **const type * const** ! En gros des constants pointeurs de constante, on comprend qu'on pourra également pointer des constantes avec. Par contre**, on ne pourra ni modifier la valeur via un déréférencement, ni le faire pointer vers autre chose**.
- Tous ces concepts de pointeurs sont différent, pour récapituler il existe : **type***, **type * const**, **type const ***, **const type * const**. Par contre, les types **type const * const** ou **const type * const** sont exactement pareil et aucun changement n'est là entre ces deux-là et entre **const type *** et **type const ***.
- Il faut retenir qu'un pointeur constant ou non pointant sur une constante ne pourra le faire que si le pointeur s'applique sur des constant.

- L'utilisation de **const_cast** agissant sur les pointeurs est également intéressante. Si on a un **const pointeur (* const)** qu'on essaye de faire pointer vers une autre valeur, même l'utilisation de **const_cast** ou **static_cast** pour le transformer en pointeur simple ne fonctionnera pas et entraineront une erreur de compilation. Pareil si on essaye de déréférencer un **pointeur de const (const *).** Erreur de compilation si on essaye d'utiliser un **static_cast**… Par contre le const_cast…
- En effet **le const_cast dans ce cas fonctionne** et permet de **modifier la valeur pointée**. Par contre, il y'a un **grand danger** car lorsqu'on fait pointer sur un const alors **la valeur est sensée être immuable**, pourtant avec le const_cast **la variable déréférencée est modifiée**, pas le constant et **pourtant l'adtesse mémoire pointée par le pointeur et celle de la constante est là même !!** Gros foutoir du coup.
- Toujours avec **const_cast**, on peut **faire assigner un pointeur de const à un pointeur de variable**. Ce qui est bizarre c'est que si on essaye de le faire en assignant le pointeur de variable avec une constante il n'est pas modifié… WTF ?
- Avec le **const_cast** la **modification du déréférencement du pointeur** est possible et modifie également la valeur, donc pas de problème ici.
- Faisons maintenant **le parallèle avec les références**, la référence a la même valeur que la variable liée, et le déréférencement donne également la même valeur. On peut également faire des **références de pointeur** qui **donneront l'adresse de la valeur pointée**, donc rien de différent. Par contre on ne peut pas pointer comme ça des références, hors d'un contexte comme les tableaux, les vector etc… Si on le fait, erreur de compilation.
- On peut également pointer un pointeur en utilisant **des pointeurs de pointeurs**. On devra faire un double déréférencement pour accéder à la variable pointée. Si on déréférence une seule fois, on tombera sur **l'adresse du pointeur** lui-même.

Pointeur de fonction (chapitre 35)

- On peut également utiliser les pointeurs comme arguments dans une fonction afin de procéder comme une fonction ayant plusieurs return
- On peut également **pointer des fonctions avec des pointeurs** ! Pour déclarer un pointeur sur une fonction : **type (*pointeur) (type arg1..).** On l'assignera ensuite ou on l'initialisera avec le nom de la fonction. Si on n'utilise pas le bon ordre des paramètres ou pas le bon type, erreur de compilation.
- Voici un exemple d'assignation : **int (* pointeur) (int) = isalnum** ;

Variable globale (chapitre 36)

- Une **variable globale** est une variable **déclaré hors d'une méthode ou une fonction**, cette variable **existera pour toutes les fonctions qui la suivent** (et pas pour toutes les fonctions qui la précèdent dans le code. En cas de tentative d'utilisation, erreur de compilation.
- Si on n'initialise pas une variable globale il faut savoir que **sa valeur sera 0 par défaut**.
- On peut déclarer dans des fonctions les **variables locales portant le même nom que la variable globale**. Quand on tentera d'accéder a une variable via ce nom dans la fonction, c'est toujours **la variable locale qui sera utilisée**. Pour accéder à la variable globale on utilisera :: avant la variable.
- Si on crée un **bloc d'instruction** dans la fonction avec **une variable locale** déclarée, ça sera **la variable locale qui sera utilisée**, si on déclare une nouvelle variable avec le même nom, c'est elle qui sera choisis a la place de la locale de la fonction et la variable globale.
- Passons au concept des **pointeurs zombies**, on explique ce concept par l'utilisation d'un **bloc d'instruction, toutes les variables utilisées dedans meurent à la fin du bloc**, mais si on **utilise un pointeur sur une variable alors lorsqu'on quitte le bloc on peut continuer à pointer sur la valeur alors qu'elle est détruite**, d'où le mot « **zombie** », car la zone pointée est encore sur la pile.

Variable statique (chapitre 37)

- Une **variable statique** est une variable qui est **créée une seule fois** et qui **ne mourra pas avant la fin du programme**, peu importe si elle est dans un bloc d'instruction ou non. Dès lors à chaque utilisation on pourra lui faire subir un traitement et elle restera dans cet état.

Priorité, associativité et évaluation (chapitre 38)

- En C++ comme dans d'autres langages, pour pouvoir avoir un contrôle sur tous les calculs qu'on fait, on a besoin de **connaître la priorité de l'opérateur** et **l'associativité** qui est une des propriétés qui définissent cette priorité. Sauf que cette associativité peut être compromise par l'**ordre d'évaluation des opérandes**, qui est l'ordre dans lequel le compilateur va lire les variables.
- Quand le compilateur évalue une instruction, il se fout de savoir qu'il y'a une multiplication en fin de calcul. Pour lui, si il y'a h() + g() * f() il verra peut-être h() avant alors que l'associativité fait qu'on doit s'occuper de la multiplication. Ces deux concepts rentrent en conflit.
- Dès lors, en fonction de **l'architecture de l'ordinateur**, certains calculs via des appels de fonctions pourront être imprévisibles car l'évaluation se fera de manières différentes.

L'inférence des types (chapitre 39 et 40)

- L'**inférence des types** est un concept décrit par la fait que certaines variables, si elles sont déclarés à l'aide de mots clés s spéciaux, adapteront leur type en fonction du contexte. Ces mots clés sont **auto** et **decltype.**
- Le mot clé **auto** peut être utilisé dans des fonctions pour **un type de retour** comme pour une **variable**. Si on l'utilise dans une variable alors elle **doit être déclarée**, car auto adaptera son type en fonction de la variable à laquelle on assigne la variable auto. Ici il y'aura donc une erreur de compilation.
- Lorsqu'on utilise **auto** comme **type de retour**, on doit malgré tout spécifier le type de retour de la variable comme ceci :
  auto f() -> type // si on ne le fais pas, erreur de compilation.

- Cela ne signifie pas que la variable de retour doit être un int, par contre **le type de retour s'adaptera à la variable** qui reçois la valeur quel que soit le type de retour, comme avec une conversion.
- Une variable **auto ne peut pas changer de type** en fonction de ce qu'elle reçoit, même si ça fonctionne, compile et affiche sans problème, le type ne change pas.
- On peut assigner une variable auto avec un immédiat et l'afficher, par contre, on ne pourra pas l'afficher en l'initialisant avec des accolades ! Avec des parenthèses ça marchera, mais pas des accolades, sinon erreur de compilation au moment où on affiche. Cela pour la raison qu'ici, auto devient un type différent, il devient une **initializer_list**, un conteneur qu'on verra par la suite mais qu'on ne peut pas afficher avec un cout tout simple.
- Parlons de la deuxième variable maintenant, il s'agit de **decltype.** Son travail est de **reconnaître le type d'une variable** ou d'un **type de retour en fonction de ce qu'on lui passe en arguments comme variable**. Ainsi si i est de type int et que j'appelle **decltype(i) j ;** il déclarera un int j.
- Si on passe les types decltype(i) avec int au **typeid** et au **is_same**, ça passe le test avec succès.
- On peut également donner **d'autres choses que des variables** en arguments pour **decltype**, ainsi donner **un appel de fonction** ou **une expression** sera tout à fait normal et accepté par le compilateur qui **découvrira le type**. Ainsi si je lui donne en paramètre un double * un int, il saura me dira que le type est double.

Conteneurs séquentiel: pair (chapitre 41)

- En deux mots, un **conteneur** est un objet qui peut contenir des variables, des objets, **pair** est un de ces objets qui peuvent en contenir.
- C'est un **conteneur** pouvant contenir **deux variables ou objets** qui peuvent être **de types différents** et qui peuvent eux même être des conteneurs.
- La déclaration se fait comme ça : pair<type1,type2> nomPair {var1,var2} ;
- Imaginons que la pair soit de type <int, double>, que se passe-t-il si on donne deux doubles dans les accolades ? **Pas d'erreur de compilation**, car les accolades sont la **liste d'initialisation** qui permet de **créer l'objet**.

- Si on veut **afficher une pair**, on ne peut pas faire un simple cout << pair. Il y'aura une erreur de compilation. Pour pouvoir afficher la pair, il faut **accéder au premier champ et au second champ**. La pair a justement **des attributs first et second pour accéder respectivement au premier et second champ de la pair**. Ainsi dans une **pair <int, double>** on accédera au int avec **pair.first** et au double avec **pair.second**.
- On pourra également **redéfinir une pair avec les accolades**, ici **toujours pas d'erreur de compilation à cause d'une conversion dégradante**, ça accepte et fait la conversion. Les seuls cas où ça ne marche pas, **c'est si la conversion est impossible**, genre donner **un string quand on demande un int**.
- On peut déclarer une pair avec les **decltype** mais on a aucune garantie que ça soit le bon type qui soit choisis pour la pair.
- On **peut initialiser ou assigner une pair avec un autre pair** en la mettant dans la liste d'initialisation ou avec un =. Cela est possible si la conversion est possible bien entendu.
- On peut également utiliser **make_pair(var1,var2)** pour **initialiser ou assigner une pair avec une nouvelle pair**. Les deux arguments de make_pair seront utilisés comme champs dans la pair. Cela marche seulement si une conversion est possible.
- Il existe également **une autre manière d'avoir accès aux champs first et second de la pair**. En effet il existe une méthode **get<> venant de <utility>** qui permet d'avoir **accès a un champ de la pair en indiquant 0 pour le premier, 1 pour le deuxième… Comme pour un tableau**.
- A noter qu'on pourrait donner **une valeur plus grande que 1**, mais pour les pair ça ne marche pas car **il n'y a que deux champ**. **get<> s'applique a d'autres conteneur** et cela sera utile a le moment-là, mais avec pair, erreur de compilation.
- Il existe également une méthode **swap** pour les pair qui permet de **changer le contenu de deux pairs** si elles ont le même type dans leurs champs, si elle n'ont pas le même type, erreur de compilation.
- Il y'a également une **surcharge d'opérateur** pour les **pair**, on pourra **comparer deux pairs si elles ont le même types en elles**, a noté que dans une comparaison > ou <, **on teste le premier champ, si les deux ne sont pas égaux, on test alors le deuxième champ pour faire son choix**.

- Pour finir, on pourra également **utiliser les pointeurs pour des objets**, à la condition qu'ils soient également **exactement de même type**. Pour procéder à l'accès aux champs, on utilisera une nouvelle syntaxe. En effet si on y réfléchit **on peut faire *pointeur.first, mais si on le fait, le compilateur pensera qu'on veut déréférencer pointeur.first**, hors, on veut juste déréférencer pointeur, et il y'aura une erreur de compilation. Cela se règle avec des parenthèses. On fait **(*pointeur).first** et cela marche. Mais il existe une écriture pour simplifier tout ça, ainsi ici **pointeur->first** permettra d'accéder au champ first de la variable pointée.
- On pourra également **utiliser auto pour savoir le type de l'objet qu'on veut pointer**.

Conteneurs séquentiel: tuples (chapitre 42)

- Un objet de type **tuple** c'est un peu comme une pair, sauf qu'on peut avoir **autant de champ dedans qu'on veut** et que l'accès ne se fera pas par first ou second.
- Comme pour les pair, il n'est pas question d'afficher juste en mettant le nom du tuple, il y'aura une erreur de compilation. On utilise ici la fonction **get<champ>(nomTuple) avec champ qui doit être un immédiat entier.** Si ce n'est pas un immédiat entier, erreur de compilation.
- La fonction get<> ne doit jamais avoir un entier plus grand que le nombre de champ -1. Sinon erreur de compilation.
- Pour **l'initialisation des tuples**, on utilise toujours les accolades et donc la **liste d'initialisation**. Pas d'erreur de compilation tant que la conversion est possible. On doit y **donner le nombre exact d'arguments** sinon erreur de compilation.
- On ne peut pas assigner un tuple avec des **types différents** à un autre, erreur de compilation. Si les types sont les mêmes, alors on peut. C'est exactement pareil pour le swap, **on peut swapper que si les types sont exactement pareil**.
- On ne peut pas **redéfinir un tuple** en lui donnant une liste d'initialisation, même si elle est correcte. Erreur de compilation. Si on lui donne un **decltype** ou **un type de tuple exactement pareil**, alors là ça marche sans problème tout comme utiliser **make_tuple** avec une liste pour redéfinir.

- Une pair n'est pas un tuple, par contre **un tuple avec deux champ peut être initialisée avec une pair**, **même si les types du tuples sont différents**, tant que la conversion est possible il le fera. **On peut également assigner une pair a un tuple sans problème**.
- Pour la **comparaison**, on peut également **comparer des tuples entre eux** comme avec les pairs. Par contre, on ne pourra **pas comparer une pair et un tuple entre eux**, erreur de compilation. De même que **les tuples qui ont un nombre de champ différent ne peuvent être comparé car les tuples peuvent être comparés que s'ils ont la même taille**.
- Parlons maintenant des **templates**. Une fonction **qui s'adapte à plusieurs type de variable** est construite avec ces templates qui sont des patrons de fonction**, c'est une alternative à la surcharge de fonction**. Au lieu d'écrire 25 fonctions pour qu'elle gère les int, les chars, les doubles… On écrit un template qui sera plus adapté.
- Pour procéder à son écriture, on écris:
  template<class Type> Type nomfonction(Type param1,Type param2…){ }
- On définira la fonction dans les accolades en **utilisant les noms des paramètres**, comme une fonction normale. A noter que **ce n'est pas la seule syntaxe possible**. Il en existe une autre pour **le cas où l'on ne connaît pas le nombre d'arguments**, on met alors **< …Args>** aux arguments ou **aux types qui ont un nombre variable d'arguments**.

Conteneurs séquentiel: initializer_list (chapitre 43)

- Une **liste d'initialisation** est une liste dans laquelle **on peut stocker plusieurs variables**. Quand on utilise les accolades pour initialiser on utilise cette même liste d'initialisation. Pareil pour **les constructeurs** d'objet où on en utilise.
- Elle se déclare comme ça : **initializer_list<type> nom {var1,var2,…}**
- On peut avoir autant de variables qu'on veut dans une liste d'initialisation.
- Pour afficher cette liste, on ne peut pas utiliser **un bête cout**, on doit faire un **foreach (for (type : liste)** ou parcourir avec des itérateur.
- La méthode size donne le nombre d'éléments.

- Si on déclare une variable **auto** initialisée avec des **accolades** et une valeur dedans, le type sera une **initializer_list,** du moins avec **gcc**, le compilateur sur Qt. Du coup si on déclare une variable **auto** avec **deux éléments** dans les accolades, erreur de compilation, **il doit en avoir une seule**. Pourquoi ? Parce que **la liste d'initialisation a deux formes**. **La liste d'initialisation directe et la liste d'initialisation par copie**. Ici on utilise la liste d'initialisation directe et elle nécessite **une seule variable**.
- Pour déclarer une liste avec **auto**, on doit utiliser la liste d'initialisation par copie, pour cela on fait : auto var = { 2,4 }, le = permet d'utiliser la **liste d'initialisation par copie**.
- Comme dis au-dessus, on utilise également **les itérateurs** pour parcourir une **liste d'initialisation**, ça sera un chapitre complet mais voici une introduction. Un itérateur est un **pointeur** qui parcourt un **conteneur** de manière particulière. En effet il faut un **itérateur** qui pointe le début du conteneur avec **la méthode ou la fonction begin()**, il en faudra également un qui pointe vers la fin avec **la fonction ou la méthode end()**.
- Pour déclarer un itérateur on initialise avec **auto** (pour le moment, on verra que par la suite il y'a plus de moyens que ça- auquel on assigne begin() ou end().
- Pour le moment, on initialisera comme ça : **liste.begin()** ou **liste.end()** ou **begin(liste)** ou **end(liste)**. Ces itérateurs ne sont pas là qu'en **lecture**, on peut les utiliser pour la **modification** des éléments.
- Vu que les itérateurs sont **des pointeurs**, si on les **affiche**, on verra une *adresse mémoire*. Il faut déréférencer pour pouvoir afficher les valeurs qu'il pointe.
- En fait, **begin()** pointe avant le premier élément, et **end()** pointe après le dernier élément, si on déréférence sur begin() ou end(), erreur de compilation.
- Pour parcourir un conteneur avec les itérateurs, on fait un for normal de type : **for (auto it = liste.begin() ; it != liste.end() ; ++it)** dans lequel on déréférence it pour utiliser les éléments.
- On utilise aussi **rbegin()** et **rend()** qui sont des **itérateur** qui parcourt à l'envers et qui parcourt de la fin vers le début si on commence avec **rbegin()** et qu'on finit avec **rend()** mais c'est une fonctionnalité du C++14 et en C++11 on aura une erreur de compilation. **.**

Conteneur séquentiel: string (chapitre 44, 45 et 46)

- On connaît le type **string**, c'est une **chaîne de caractère** qui permet de faire les affichages qu'on fait depuis longtemps, mais il faut savoir que **string est un alias d'une autre classe nommée basic_string<char>**, quand on l'utilise sous l'alias, on se rend compte que c'est l'un des seuls qui utilise pas de **chevrons**.
- La déclaration se fait donc comme cela : **string nom{ '' hkjhl''}**
- Cette classe string comporte de nombreuses **méthodes** intéressantes. Parlons d'abord de **size()** et de **length()**, ces deux méthodes retournent le nombre de caractère de la chaîne. La méthode **capacity()** donne le nombre de caractère possible avec **l'espace mémoire** réservé pour la chaîne. Donc si on procède à ce genre d'initialisation, **capacity(), size()** et **length()** donneront la même résultat. La méthode **max_size()** donnera le nombre de caractère possible avec la mémoire disponible sur le pc.
- Une variable de type **string** c'est comme un **tableau**, on peut y accéder avec **l'opérateur []** pour accéder à chaque caractère. Par exemple dans la chaîne, «salut», **string[3] permet d'accéder à 'u'**. Il existe également la méthode **at(index)** pour accéder comme avec [], sauf qu'**avec [] il n'y a aucune vérif.**, alors qu'**avec at() il y'en a une**. Si on dépasse **une exception se lève** et on a une erreur à l'exécution. Si on donne une valeur flag avec [] genre 100000 aussi il y'en aura une.
- Pour **afficher un string**, pas besoin de foreach ou itérateurs, juste un **cout**.
- Pour ne pas avoir une **erreur a l'exécution** si on dépasse on utilise les **exception** avec les **try-catch**, on met ça dans un try et si ça dépasse on affiche le message de l'exception : **try {…} catch(exception e){…)**
- On peut également **réserver des caractère** dans un string initialisé ou non avec la méthode **reserve(nombreCaractère)** pour réserver un nombre de caractère, avec cela, le résultat de **capacity()** deviendra égal a **nombreCaractère**. On réservera plus de mémoire pour le string avec ça.
- Avec **clear()**, on effacera tout ce qu'il y'a dans le **string**, **la capacité** restera la même, par contre, **on effacera tous les caractère** et la chaîne sera **vide**.

- Parlons maintenant de la **réallocation automatique**. En effet, lorsqu'on veut met plusieurs chaîne les unes à la suite des autres ou qu'on utilise **l'opérateur +**, on concatène les chaîne. On **rassemble la mémoire** et on crée un nouveau string qui peut contenir toute cette mémoire. Et ça se fera sans problème, même si on a réservé moins de caractères.
- Parlons maintenant des **raw string,** avant d'en parler il faut parler des **caractères d'échappement**. En effet, quand on affiche **'' \n''** on n'affiche pas vraiment cela. En réalité on passe à la ligne et cela existe pour de nombreux caractères. **\\ permet d'afficher \, \t permet de faire une tabulation. \'' permet d'afficher ''**. Cela ne marche pas avec tous les caractères, en effet si \x n'existe pas et qu'on le met dans une chaîne, on aura une erreur de compilation. Sachant que '' et \ peuvent aussi être utilisé dans les **chaînes** ou on veut vraiment mettre \t dans une chaine, ça ne l'affichera pas et les **raw string** résolvent ce problème.
- Pour utiliser la **raw string**, on utilise deux écritures. Soit **R''(…)''** , soit **R''xyz(…)xyz''**, tous cela dans une chaine de caractère avec des caractères d'échappement à la place des …
- Le seul caractère qui pose problème dans une **raw string** c'est '' car les deux '' déterminent le **début** et la **fin**. Du coup, on en met un au milieu sans \ devant, erreur de compilation. C'est là que la deuxième écriture de raw string intervient. En effet **R''xyz(…)xyz'' permet d'avoir une séquence de caractère qui ici est xyz (qui peut être juste x, ou juste y ou xy, du moment que c'est la même au début et à la fin…** Ici pas d'erreur de compilation.
- On peut également **concaténer** avec un cout en faisais cout << ''jbgbg'' ''jkhkl'' 'hiohjih'' << endl, ça concatènera les 3 ensemble.
- Il existe d'autres méthodes comme la méthode **find(char)** qui permet de **retourner la position de la première occurrence du caractère recherché**. Si le caractère n'est pas trouvé alors la fonction **retournera un attribut nommé npos** qui veut dire que **le caractère n'a pas été trouvé** dans la string. Il existe également la méthode **substr()** qui permet **de découper la chaine et d'en prendre une partie**. Si on donne **un paramètre** ça prend de la position en param jusqu'à la fin de la chaîne. Si on en a deux, **comme départ on prend le premier param** et on prend le nombre de carac donné par le 2eme param. Ex : substr(3,5) => 5 caractère à prendre

- La méthode **empty()** permet de déterminer si **la chaîne est vide** ou pas. Elle retourne vrai ou faux. La méthode **replace()** permet de remplacer **une partie d'un string par une autre donnée en paramètre**. Elle prend en paramètre **la position de départ à partir de laquelle on remplace**, ensuite **le nombre de caractère qu'on remplace** et **pour finir par quoi on remplace**.
- On peut également convertir des **variables numériques en string** ou l'inverse. Pour la conversion **numérique => string**, on utilise le **to_string** avec la variable. Cette fonction marche avec tous **les types primitif numérique qu'on connaît (int, double, long, unsigned, float**… A noter que **si l'argument est de type char (unsigned ou signed) il est promu en int ou unsigned**.
- On peut donc également **faire l'inverse** et passer **de string a numérique**. On passe alors via **des méthodes spéciales pour chaque type** on retrouvera **stoi** pour les **entiers**, **stod** pour les **double**, **stol** pour les **long** etc… On lui donne en paramètre **la chaine de caractère**, **un pointeur** (qui peut être nullptr) et la base qui sera 10 si on ne met pas ce param. Ces fonctions sont très sensibles si puis-je dire, parce qu'elles peuvent vite mal tourner contrairement à l'autre conversion, on a donc l'habitude de les mettre dans un **try-catch sinon <u>erreur a l'exécution</u>**.

<u>Conteneurs séquentiel: array (chapitre 47)</u>

- Le conteneur **array** lui, représente presque les tableaux vu en première en java, assez facile à comprendre car très simple.
- L'instanciation se fait en ayant au préalable fait **#include array** sinon <u>erreur de compilation.</u> Ensuite, il faut écrire <u>array<type,taille> nom {var..}</u>
- Ce genre de conteneurs ne peut contenir qu'un seul type d'éléments.
- Pour l'accès aux éléments il se fait également avec [] ou la méthode **at().** Avec les mêmes caractéristiques.
- Les méthodes **front()** et **back()** permettent respectivement d'accéder au **premier** et au **dernier élément** de l'array.
- La fonction **get<>** est également utilisable ici afin d'accéder a un élément du tableau. Cette fonction a également une vérification tout comme at()

- Parlons maintenant d'un nouveau **principe**, celui du **constricteur par copie**. Ce principe consiste à utiliser un constructeur afin de **créer un objet qui a les mêmes caractéristiques qu'un objet déjà créer**. Pour cela en général, on utilise un constructeur créer pour cela, ce constructeur existe déjà pour array.
- Le seul problème ici c'est qu'il ne faut **pas utiliser les accolades**, sinon on nous dira qu'on ne peut pas convertir un **array** en un autre même si on utilise **decltype**. Sinon erreur de compilation, mais avec **auto** ça marche et aussi avec des parenthèses.
- Dès lors, si on modifie l'objet crée par le **constructeur par copie**, l'original ne sera pas modifié. Par contre, si on aurait fait **tab1 = tab2** alors les deux seraient liés et une modification dans l'un aurait entrainé une modification dans l'autre.
- Si on déclare un tableau sans l'initialiser, on ne sait pas ce qu'il y'a dedans.
- Il existe une méthode **fill(variable)** qui permet de remplir tout l'**array** avec une valeur. Ainsi **array.fill(0)** remplira tout mon tableau de 0.
- Pour les **comparaisons**, c'est également le même système qu'avant avec les **différents opérateurs** qui vérifient le premier élément, puis qui jugent sur la suite si les deux premiers ne permettent pas de faire un choix.
- On doit initialiser la taille du tableau avec un immédiat ou une valeur connue à la compilation, sinon, erreur de compilation.
- Les **expressions constexpr** faisaient appel a des **fonctions constexpr** doivent être initialisés avec des parenthèses, sinon erreur de compilation.
- On ne peut **pas comparer ni assigner** deux tableaux les uns aux autres si les types ne sont pas exactement pareils.
- On ne peut pas initialiser un tableau **avec + d'éléments** qu'ils ne peuvent en transporter.

Conteneur séquentiel: vector (chapitre 48)

- Un **vector** est un tableau assez spécial dans le sens où **la taille peut varier** pendant le programme et qu'il n'a pas besoin d'être **initialisé**.
- Il y'a beaucoup de manière de l'**initialiser**. On peut lui donner une **liste d'initialisation** avec tous les variables. On peut également lui donner **sa taille entre parenthèse**, elle peut être de n'importe quel type cette taille sera toujours convertie en **int**. On peut également lui donner sa **taille** et un entier qui sera dans toutes les cases du **vector,** toujours entre **parenthèse**, car si on met des **accolades**, on initialise avec deux variables et la **liste d'initialisation**. On peut également **d'initialiser avec un autre vecteur**, cette initialisation se fait via les **accolades** ou les **parenthèses**. Une autre initialisation peut se faire avec les **itérateurs** sur un vector en donnant **begin**() et **end**().
- Dans la **liste d'initialisation** d'un vecteur, on doit mettre un seul type de variable, sinon, erreur de compilation.
- On ne peut pas initialiser un **vector** avec un vector qui n'est pas du même type.
- Pour utiliser vector, il faut obligatoirement faire **#include <vector>** sinon, erreur de compilation.
- Les **types** dans un vector resteront les même quoiqu'il arrive, même si on **multiplie par un double un vector de int**, il n'y aura **aucune modification du type** et il y'aura conversion du double en int dans cet exemple.
- On peut assigner un **vector** a un autre, à conditions qu'ils soient du **même type**, sinon erreur de compilation.
- On peut **modifier les valeurs** d'un **vector** en utilisant la méthode **assign**(), avec cette méthode on peut procéder comme avec le **constructeur pour l'initialisation**, en donnant les bons paramètres.
- L'accès aux éléments se fait via **at**() , [], **back**() pour l'accès au dernier élément et **front**() pour l'accès au premier élément.
- Il existe également **push_back**() ou **pop_back**(). La méthode **push_back**() permet de rajouter des éléments à la fin du **vector**. La méthode **pop_back**() permet de retirer des éléments à la fin du **vector**. Elles prennent un seul argument en paramètre

- La méthode **resize**() permet de manipuler la taille du **vector**, en remplaçant si besoin les nouvelles variable par le deuxième paramètre.
- La méthode **clear**() permet d'effacer tout ce qu'il y'a dans le vector.
- Il existe également les méthodes **swap**(), **insert**() et **erase**(), **emplace_back**() (pour les construction d'objet dans les **vector**() ou le **shrink_to_fit**() pour annuler une réserve de mémoire.
- Pour la comparaison de différents **vector**, on compare toujours sur le **premier élément** et si il ne permet pas de se décider alors **on passe au suivant, même si l'un des vector est plus petit que l'autre**. A Noter qu'on ne peut pas comparer deux vector de types différents, sinon erreur de compilation.

Conteneur séquentiel : deque (chapitre 49)

- Un **deque** est également un conteneur à taille dynamique, il y'a assez peu de choses à dire, néanmoins le deque a comme **spécialité de pouvoir rajouter ou enlever des éléments au début et à la fin**, un peu comme une liste chainée.
- La **construction** se fait comme vector, à la différence qu'on peut **initialiser** un deque avec un vector.
- **L'accès aux éléments,** la **taille** et la **comparaison** entre deque est pareille que pour vector.
- On utilise dorénavant deux nouvelles méthodes, **push_front**() et **pop_front**() pour enlever ou ajouter des éléments en début de deque.
- Il existe d'autre conteneurs séquentiel (car l'accès et séquentiel via l'index et l'itérateur) comme les **forward_list**, les **list**, les **stack**, les **queue**… Mais il existe d'autres conteneurs nommé les conteneurs associatifs.

Conteneur associatif : set (chapitre 50)

- Un set est un conteneur associatif. Ce sont des conteneurs qui ont une **clé.** Il en existe deux types : les conteneurs associatifs à clé **unique** comme **set** et **map**, qui **refuseront** les **doublons** et les conteneurs à clé **multiples** qui accepteront ces doublons comme **multiset** et **multimap**. Ces deux types de conteneurs ont leurs avantages.

- La construction de l'objet se fait comme ceci : **set<type> {var1, var2,…},** si on donne plusieurs fois la même **valeur** dans la liste, elle ne sera pas rajoutée dans le set. On peut également initialiser en utilisant les itérateurs et en donnant. On ne peut pas initialiser un set avec un set d'un type différent, il y'aura une erreur de compilation. Avec un set du même type, c'est possible. D'habitude, lors de la **construction**, on doit mettre deux paramètres entre <>, il y'a le type mais aussi un objet de **comparaison** pour choisir l'ordre de tri, la construction alternative se fait comme ceci : **set<type,comparateur> nom {var1, var2,…}**
- On doit obligatoirement procéder à l'include de set via **#include <set>**, sinon, erreur de compilation.
- Mais qu'est-ce qu'un **comparateur** ? En fait ce sont des objets que nous avons déjà croisé comme **greater<type>** qui permettent de trier un conteneur avec un **ordre** particulier. Par défaut, le **set** utilise un **comparateur** qui trie dans l'ordre croissant, mais rien ne nous empêche d'utiliser un autre comparateur, à la condition de le préciser à la **déclaration**. Ces **objets**, qui peuvent aussi être des **struct,** prennent généralement deux paramètres, qui sont **les deux valeurs à comparer** et retournent un booléen qui dis si celui de droite est « **plus grand** » que celui de gauche dans la comparaison choisie. On peut définir nos propres ordre en les créant nous-même par exemple en faisant une **struct** avec un **operator**() définis dedans qui trie dans l'ordre qu'on choisis… ou via une **expression lambda** (vu par après)
- On peut également **assigner** un set à un autre, à la condition qu'ils aient le même **type** ET qu'ils aient le même **objet de comparaison**.
- Il existe plusieurs **méthodes** pour les **set**(), notamment la méthode **insert**() qui permet d'insérer **une ou plusieurs valeur**, notamment avec les **itérateurs** ou avec **une seule variable insérée**. Cette méthode renvoie une **pair** qui contiens un **itérateur** sur la **valeur** qu'on souhaite ajouter, et un **booléen** qui dis si la valeur est ajoutée ou non dans le set. On peut ajouter plusieurs valeur en mettant une liste **d'initialisation** (avec { }) qui comprends plusieurs variable
- La méthode **count(valeur)** permet de compter le nombre **d'occurrence** de valeur dans le **set**, vu que c'est un **conteneur à clé unique**, le résultat sera soit 0, soit 1.

- La méthode **find(valeur)** cherche une valeur dans le **set**, et si elle la trouve, elle revoie un pointeur sur la variable trouvée dans le **set**.
- La méthode **erase(iterateur)** permet d'effacer le contenu d'un set en partant de **l'itérateur** donné en **paramètre** jusqu'à la fin du conteneur. Une alternative de cette fonction est la surcharge **erase(valeur)** qui permet de faire **un find(valeur) et un erase en même temps** si elle est trouvée, sinon aucun changement. Pour finir la méthode **clear()** quant a elle, elle **supprime** tout ce qu'il y'a dans ce **set**.

Conteneur associatif : multiset (chapitre 51)

- Un **multiset**, comme dis précédemment est un conteneur **à clé multiple**. Comme les set, il ne contiens que la clé et pas de valeur contrairement aux map.
- Si on essaye de rajouter plusieurs fois la **même valeur** dans le **multiset**, elles seront ajoutées contrairement au **set**.
- La construction et l'initialisation d'un **multiset** se fait comme cela : **multiset<type> nom {val1,val2,..}**
- Pour utiliser le multiset, il faut faire **#include <set>**
- Il existe les mêmes méthodes qu'on a pour **set** dans **multiset**. Donc **find**(), **erase**(), **count**()… sont encore applicable ici.
- La fonction **distance(begin(),end())** permet de dire la distance entre la première occurrence d'une valeur pointée par un itérateur et sa dernière occurrence.
- La méthode **equal_range(valeur)** renvoie une pair de deux itérateur, le premier équivalent à la dernière clé qui n'est pas plus petite que la valeur, et le deuxième au premier élément plus grand que la clé. Il combine deux méthodes. La méthode **lower_bound(valeur)** pour la partie sur **l'itérateur** par plus petit que la valeur et la méthode **upper_band(valeur)** pour le premier élément plus grand que la clé. On peut également donner une valeur qui n'est pas une clé dans le **multiset**.

Conteneur associatif : map (chapitre 52)

- Une map est également un conteneur associatif, mais ce qui le différencie de set et multiset c'est qu'il a **une clé et une valeur**. Vu qu'il a une clé unique, on ne retrouvera au max 1 fois chaque valeur.

- Pour utiliser les map, il faut obligatoirement inclure <map> via la directive **#include <map>.**
- La construction se fait comme ceci : **map<typeClé,typeValeur> nom** ; On a pas besoin d'initialiser avec une liste d'initialisation. La map est vide à ce moment-là et on peut vérifier ça avec la méthode **empty**() qui était également une fonction de **set**, d'ailleurs tout ce qui était vrai pour set au niveau des méthodes, l'est pour **map**. Il faut savoir que les **map** sont composés d'objet pair en son sein. Avec comme **premier champ la clé** et comme **deuxième champ la valeur** à laquelle la clé est liée.
- Du coup, lors de **l'insertion**, pour rajouter une valeur dans la **map** via la méthode insert, on créera une **pair** avec les mêmes types que celle de la clé et de la valeur. On doit également créer un **itérateur** sur cette pair. Cela marchera également si on utilisait **insert(make_pair(…))**. On pourrait également insérer avec une liste d'initialisation qui comporte **des mini liste** pour chaque pair comme ça : **mappy.insert({{clé,val},{..}})**
- Pour accéder à une valeur de la map, on accède via la clé et la méthode **at(clé)**. On peut également procéder comme cela **map[clé].** Si on veut modifier une valeur, alors on utilise **map[clé] = nouvelleValeur.** Si on veut changer la clé, on supprime l'élément de la map et on en crée un nouveau. Pour supprimer on utilise également la méthode **find(clé)** qui renvoie un itérateur, on fait ensuite **erase(itérateur)** pour supprimer cette clé. Pour créer la nouvelle clé on fait comme la **modification**, sauf qu'on donne une nouvelle clé.
- Toutes les méthodes comme **count(), erase(), find()**… sont comme set et multiset.

<u>Conteneur associatif : multimap (chapitre 53)</u>

- Multimap est a map ce que multiset est a set. C'est-à-dire que c'est un conteneur associatif a clé multiples, on pourra retrouver plusieurs valeurs pour une même clé. On doit également obligatoirement faire cet include : **#include <map>**
- La construction se fait comme ça : **multimap<typeClé,typeValeur> nom ;**
- Toutes les méthodes comme empty(), erase(), count()… Sont comme set, map.

- La méthode **insert**() s'utilise comme **map**, c'est-à-dire qu'on peut créer une pair qu'on va insérer avec insert, on peut également utiliser **insert(make_pair(clé, valeur)),** ou une liste d'initialisation comme ceci : **mappy.insert({ {clé1,val1} , {clé2,val2} , {…} }).**
- Pour **accéder aux éléments [] et at() n'existent pas pour multimap**, ils n'existaient pas non plus pour set et multiset. Pour ajouter dans une multimap, on utilise soit **insert**(), soit **emplace**(), soit **emplace_hint**(). Pour supprimer, on utilise **erase**() et **clear**(). On peut également swapper deux map avec swap.
- Si on veut changer une valeur, il faut éliminer la pair et créer avec une nouvelle valeur, pareil **si on veut changer la clé, on supprime l'élément et on le recrée.**
- La méthode **find**() existe également pour les map, on accèdera a toutes les clé de la map qui ont cette valeur. On pourra également utiliser **equal_range(clé)** pour pointer sur toutes les valeurs d'une clé. Ces deux méthodes ont la même finalité.

Fonction de hash (chapitre 54)

- Une **fonction de hash** consiste à envoyer une **donnée** en paramètre et qu'elle ait un **identifiant unique** en sortie.
- La fonction de hash en C++ s'écrit comme ça **hash<typeAHash> nom** ;
- Ce hash fonctionnera que pour des **objets** d'un **type**, mais une conversion est possible et on peut envoyer tout type de **donnée primitif**.
- Ainsi pour les objets pour lequel ça marche, il faudra créer un type comme vu au-dessus avec le type de l'objet.
- Avec certains objets comme des **conteneurs** (qui contiennent eux même des objets ou **variable** et peuvent être définis selon plusieurs types genre **vector<int>** ou **vector<double>**) il y'aura erreur de compilation si on essaye de créer une fonction de hash pour eux.
- Pour le reste, l'utilisation du hash se fait comme cela : **nomHash(var)** qui renverra une variable entière qui sera cet identifiant.

Conteneur associatif : unordered multimap (chapitre 55)

- Comme le dit le nom, c'est une **multimap** qui n'est pas triée, elle sera affichée selon l'ordre dans lequel on rajoute chaque élément.

- Tout comme la **multimap**, on a également affaire ici à un conteneur associatif a clé multiple.
- Pour faire une recherche par clé, on peut toujours utiliser la méthode **equal_range(clé)** qui nous fera pointer sur le premier élément de **l'unordred_multimap** qui est égal à la clé, si il existe. On peut également faire un foreach qui parcourt, et pour accéder à la valeur ou à la clé, on utilise second ou first vu que l'**unordred_multimap** est également composée de pair.
- La construction de l'objet se fait comme ça : **unordered_multimap<clé,valeur> nom** { pair1, pair2, etc…} ;
- Toutes les méthodes qui marchaient sur multimap marchent également sur ce conteneur.

Les itérateurs (chapitre 56)

- Jusqu'à présent, les **itérateurs** n'ont pas été vu en profondeur, un petit **rappel** s'impose. Un itérateur est un **pointeur** qui parcourt un **conteneur**. Nous avons vu les méthodes **begin**() et **end**() qui permettaient d'accéder a tout le conteneur si on commençais à begin() et qu'on finissais à end (pour rappel, si on tentais d'accéder a **end**() en le **déréférençant**, erreur de compilation). Si on voulait **déréférencer** un **attribut** ou une **méthode** d'un objet sur lequel l'itérateur pointais on utilisais la syntaxe suivante **pointeur->méthode().** Pour avancer dans le conteneur, on incrémentait l'itérateur. Il existe en fait plusieurs types **d'iterateur** et celui qu'on a vu jusqu'à présent est **l'input iterator**. Cet itérateur est un itérateur constant, lors de la déclaration on écrira donc **const_iterator**.
- Un autre type d'iterateur est le **forward iterator.** Il produit exactement les mêmes résultats que l'input iterator si on veut parcourir la conteneur en utilisant l'incrémentation, par contre, impossible de décrémenter, il va seulement vers l'avant, on ne pourra pas partir de end() vers begin() sinon erreur de compilation. Cet itérateur est un **input iterator** si on le déclare comme **const_iterator**.
- Le **bidirectional iterator** est un **forward iterator** qui peut aller dans les deux sens, on peut l'incrémenter ou le décrémenter, ainsi si on veut on pourra aller de la fin vers le début ou l'inverse. Cet itérateur est un **input iterator** si on le déclare comme **const_iterator**.

- Un **random access iterator** est un **bidirectionnal iterator** pouvant être utilisé pour accéder à n'importe quel élément sans passer par un chemin obligatoire comme dans les **bidirectionnal iterator** qui ne pouvaient pas faire autre chose que des **incrémentation** ou des **décrémentation**. Avec cet iteréteur on pourra accéder par exemple en faisant ***(pointeur-2)** ou **pointeur[2]** (qui signifie ***(pointeur + 2)** ou **comparer** deux itérateur de ce type en faisant **pointeur > pointeur2.** Cet itérateur est un **input iterator** si on le déclare comme **const_iterator**.
- Jusqu'à présent, on a vu que des itérateurs constants, si on les déclare **iterator** tout simplement alors ils deviennent aussi des **output iterator**. C'est-à-dire qu'on pourra les modifier lors du déréférencement, ce que les **input iterator** ne pouvaient pas faire.
- Il existe également les **reverse iterator** qu'on retrouvera grâce aux fonctions **rend**() et **rbegin**() qui permettent de parcourir le conteneur a l'envers en commençant pourtant à **rbegin** et en finissant a **rend**, j'ai déjà expliqué le principe en haut.
- Pour savoir le nombre d'éléments entre deux iterateurs on utilisais la méthode **distance(it1,it2)** décrite dans les chapitres précédents, cela vaut pour tous les iterator, mais pour les **random access iterator** on peut également faire **it2-it1** pour savoir le **nombre d'éléments qui les sépare**
- La question qu'on doit se poser c'est comment déclarer un **forward iterator**, un **random access iterator, un bidirectionnal iterator**, en fait… On ne peut pas choisir. Cela se fait automatiquement selon le **conteneur** sur lequel on déclare **l'iterator**. Ainsi pour une **liste**, il sera **bidirectionnel**, pour un **array** ça sera un **random access iterator,** pour **une liste chaînée** ça sera un **forward iterator**. Pour déclarer un itérator sur un conteneur on fait **nomConteneur<> ::iterator nomIt = …**

Invalidation d'itérateur (chapitre 57)

- Si on fait pointer un iterator sur un conteneur, il faut savoir qu'on ne pourra peut-être pas y avoir accès juste après avoir **insérer**, **supprimer** un élément ou **changer la taille du conteneur**. En effet en fonction du conteneur, l'itérateur deviendra **invalide** et on ne pourra plus y accéder sans erreur à l'exécution. En l'occurrence ici, on parlera du **vector** pour lequel ces problèmes sont réels.

- Pour insérer des variable grâce a insert(), il faut utiliser un itérateur qui reçois le résultat de insert(), ou on peut utiliser la fonction inserter sur un itérator afin de créer un insert_iterator, un itérateur fait pour insérer, mais attention a ne jamais le référencer, sinon erreur de compilation, car c'est un output itérator, il ne peut pas être vu en lecture.

Algorithme lambda (chapitre 58)

- Les expressions lambda sont des moyens pour définir ce qu'on appelle des algorithmes, les lignes de code écrit dans le corps de la lambda expression seront utilisées par cet algorithme pour produire des effets intéressant. Nous avons déjà vu le cas avec le tri avec le modulo 3, grâce à la fonction sort, nous avons pu définir un ordre de tri nous-même en utilisant un algorithme. Ces algorithmes sont définis dans **#include <algorithm>**
- Par exemple ce bout de code nous permettra de générer un nombre égal à ce qu'on va définir dans le corps de la lambda expression, tout à grâce à la fonction generate.

```
generate(vi.begin(), vi.end(), []()
{
    return nvs::random_integer(1, 6);
}); // http://en.cppreference.com/w/cpp/algorithm/generate
```

- Le troisième paramètre qui comporte le []() permet de faire appel à la lambda expression qu'on définis tous de suite après entre {}.
- On peut définir sa propre expression lambda en créant une variable auto et en faisant **auto nom = [](type variable passée) -> type de retour { ..} ;** qu'on appellera en faisant nom(param) comme une fonction alors que c'est une variable, voilà, on vient de **stocker un lambda**
- Il existe d'autres algorithme comme **all_of, any_of et non_of** qui permettent de vérifier si des éléments d'un conteneur respectent une condition de la lambda. L'algorithme **count_if** permet de compter ceux qui correspondent a la lambda. L'algorithme **remove_if** permet de les supprimer s'ils respectent la condition. Il y'a plein d'autres algorithmes comme **distance()** qu'on connais déjà. On vois d'autres algorithmes comme copy, reverse, minmax_element etc… Plein d'algorithmes avec plein de possibilités différentes.

- Une variable qui stocke une lambda peut également capturer les variables qui répondent à la condition qu'on lui envoie, pour faire ça, on met dans les [] la variable déclarée dans la lambda expression qu'on veut capturer. On peut également mettre une référence ou un objet en mettant &variable ou **this**. L'utilité sera par exemple d'afficher ce qu'il capture et la réponse de cet lambda expression via l'algorithme. On peut également tout capturer en mettant [=] ou [&]

Les classes (chapitre 59)

- Une classe définis le concept d'orienté objet en C++. On les décrits dans le fichier header avec le mot clé **class** suivi du **nom** de la classe, par exemple Foo.
- Dans l'OO, on retrouve également le concept d'encapsulation, en java on devait mettre **private**, **public** ou **protected**. En C++, on utilise également ces mots clés mais si on ne met aucun mot clé, alors par défaut tout est **private**. Pour déclarer quelque chose comme public, on le met suivi de : et toutes les déclarations qui le suiveront seront public, sauf si on met un autre mot clé par la suite. Exemple : **public : … private : …**
- Une bonne pratique en C++ c'est de faire suivre le nom des attributs par _ exemple **std::string nom_**;
- Dans la classe on ne définis que les attributs et les entêtes de méthodes qui seront définis dans le **.cpp**, sauf si ce sont des méthodes **inline.** Cela vaut pour tout, même les constructeurs.
- Il est désormais important de définir la différence entre **méthodes** et **fonctions**. Les **méthodes** sont des fonctions qu'on utilise sur un objet en y accédant via l'opérateur «.», les fonctions ne sont pas utilisable sur un objet via cet opérateur mais via l'appel de fonction.
- On peut définir autant de constructeur qu'on veut, il faut savoir que par défaut il existe un constructeur gratuit qui ne prend aucun paramètre et qui cesse d'exister quand on définit un autre constructeur. Malgré tout, on peut le recréer en définissant une entête de constructeur et en faisant **Constructeur() = default**.

- Continuons avec les constructeurs. Le constructeur par défaut n'a pas besoin d'être implémenté, mais les autres oui. Pour initialiser les attributs, on a deux choix. On peut utiliser le **corps du constructeur** ou sa **liste d'initialisation**. Pour le corps du constructeur c'est comme on a toujours fait, c'est-à-dire la définition entre les deux accolades. Pour la liste d'initialisation on met donc le nom du constructeur, suivi de : puis chaque attribut avec une valeur d'initialisation. Comme ceci :

```
Foo::Foo() : Foo{ 1, 4.3 }  // délégation de construction
{ }

// liste d'initialisation de constructeur : toujours privilégier
Foo::Foo(int i, double d) : i_ { i }, d_ { check(d) }
{
    check();
}

Foo::Foo(int i, double d, unsigned su) : Foo { i, d } //, su_ { su }
{
    su_ = su;   // accès attribut statique dans méthode non statique
}
```

- On peut également utiliser les deux manières en même temps, comme dans le troisième exemple de l'image au-dessus. Et même appeler un constructeur définis dans le code comme le premier exemple.
- Si on a définis une méthode ou fonction statique dans le **header**, il ne faut pas remettre le **static** devant lors de l'implémentation, sinon, erreur de compilation.
- Dans le main, lorsqu'on crée un objet sans l'initialiser, on appelle le constructeur par défaut. Ainsi **Foo fo;** appellera **Foo().** C'est une manière implicite d'écrire **Foo fo{};** On peut également redéfinir un objet créer en faisant **objet = Foo{…}** ou pour le constructeur par défaut **objet = {}** suffira. En effet ceci fonctionne car **l'opérateur** = est **surchargé** par défaut pour chaque **objet**.
- Pour les membres **private** de la classe, on ne pourra pas y accéder en dehors via un **appel**, mais seulement via un **accesseur** en **lecture** (**get**) ou en **écriture** (**set**). Et pour y accéder aux méthodes static, il faudra déréférencer via **nomDeClasse::** avant chaque appel. Par exemple **Foo::setO()**
- Pour accéder à l'objet lui-même dans une méthode, on utilise **this**, mais c'est un pointeur du coup pour déréférencer on utilisera **this->…**

- Pour finir, on peut initialiser les attributs dans la classe elle-même, sauf les attributs statique, qu'on devra initialiser dans le constructeur ou autre part. En tout cas pas dans le corps de la class, sinon erreur de compilation.
- On a l'habitude de déclarer **const** les fonctions comme les getters ou les **to_string** car ils ne modifient pas l'objet, mais si on l'enlève ça change rien.
- Finissons-en avec les fonctions **amies**. On les déclare avec le mot clé **friend**, ces fonctions auront accès aux membres **privés** de la classe tout en étant pas dans la classe. Et c'est ça l'intérêt, c'est qu'une fonction rien à voir puisse avoir accès aux attributs **privés** tranquille. On pourra avoir accès à ces attributs **privés** via le déréférencement **nomClasse::.**

Les constructeurs par défaut (chapitre 60)

- Un constructeur par défaut est un constructeur sans paramètres qui existe lorsque aucun constructeur n'est définit. Mais pas seulement, on peut définir soi-même un constructeur par défaut via le mot clé default comme vu au-dessus.
- Le constructeur par défaut n'existe plus quand un constructeur est définis sois même. Si on essaye d'appeler ce constructeur (qui est sans paramètre) il y'aura une erreur de compilation.
- Un constructeur par défaut peut être implicite ou explicite, comme vu au-dessus, l'implicite est créé sans qu'on ne fasse rien tandis que l'explicite est créer quand on le fait soi-même via le mot clé **default**.
- Pour créer un **constructeur explicite** on crée un constructeur sans paramètre et on fait **Constructeur()= default ;** dans la définition de la classe, ce constructeur doit être sans paramètre.
- Pour donner une valeur par défaut aux attributs dans un objet construit par un constructeur par défaut, on peut initialiser les **attributs** dans la classe. Pour cela il faut bien faire attention aux **narrowing**. Pour les éviter, on n'utilise que le **«=»,** les **parenthèses** ne compileront pas !
- En général, les constructeur par défaut ne servent à rien car l'objet est créé dans un état indéfini, ça peut servir que si on donne une valeur par défaut aux attributs, dans ce cas, la valeur sera toujours garantie.

- On ne pourra pas créer un objet en lui donnant un paramètre si on a qu'un constructeur par défaut.
- Si on créer juste un objet, on aura une valeur rien a voir, si on le crée et on l'initialise, on aura la valeur 0 pour les attribut ainsi : **Foo a1** ; et **Foo a2{}** n'ont pas le même résultat. Par contre ça sera la même chose si on initialise les attributs avant. Cela marche pour les constructeurs implicites ou explicites.

Surcharge de constructeur (chapitre 61)

- Les règles pour le choix de fonction en fonction du type valent toujours avec donc la correspondance exacte puis la conversion numérique qui prime.
- Il existe une technique nommée la délégation de construction qui permet d'appeler un autre constructeur quand on définit un constructeur. Ceci permet de faire le travail proprement. L'exemple est au-dessus dans le chapitre 60.
- On sera confronté aux mêmes problèmes que pour les surcharges, si une conversion vers deux types est possible on aura des ambiguïtés. De même que si on donne deux paramètres ils peuvent créer une conversion dégradante, on pourra l'éviter avec les parenthèses au moment de créer les objets (ça se fait dans le main, pas dans la classe).
- On n'aura pas de problème d'ambigüité si on a un seul constructeur, bien entendu.

Constructeur et valeurs par défaut des attributs (chapitre 62)

- Ce chapitre est assez bizarre dans le sens où il parle des attributs statiques et constexpr qu'on pourra utiliser ou non.
- Ainsi pour déclarer un attribut static qu'on va initialiser on aura deux choix, soit c'est un nombre décimal, soit un nombre entier. Si c'est un nombre entier, on le déclarera **static const** alors que si c'est un nombre à virgule on le déclarera **static constexpr**. Aucune idée pourquoi, mais c'est comme ça. Cela ne vaut que pour **les attributs qu'on initialisera dans la classe.** Sinon, on pourra utiliser **static** seul.
- On sera obligé d'initialiser un attribut **const** dans la classe si il y'a un constructeur par défaut

- Il faut faire attention avec les parenthèses dans la définition de la classe. Ainsi on ne pourra pas déclarer un vector avec les parenthèses, car il le prendra pour un **prototype**.
- On ne pourra pas accéder à un attribut non statique sans construire un objet. Si on essaye de le faire, erreur de compilation. On pourra accéder aux attributs statique en faisais **Objet::nomattribut ;**

Constructeur explicite (chapitre 63)

- J'ai parlé d'**implicite** et d'**explicite** au-dessus, mais ce qui va suivre est le véritable sens du mot clé **explicit** en C++, pour cela, on doit définir ce qu'est un **appel implicite** d'un constructeur et un **appel explicite** d'un constructeur.
- Un appel **implicite** c'est lorsqu'on utilise le **=** avec une **valeur** sur un **objet**, on appelle le **constructeur** en le faisant. Ainsi **objet = 4** ou alors **objet = {4}** sont des appels implicite du constructeur, les appels explicites sont les appels différent de ceux-ci comme **objet{}** ou **objet**;.
- Le mot clé **explicit** est utilisé dans la définition de la classe et est réservé aux constructeurs, lorsqu'on déclare un **constructeur explicite**, il nous est plus permis de faire des **appels implicite**.
- On le déclare comme ceci : **explicit Constructeur() ;**
- Dès qu'on tentera de faire un appel implicite à ce constructeur, on aura une erreur de compilation.
- Quand on a des constructeur explicite, et qu'on fais un appelle implicite, les problèmes **d'ambiguité** peuvent disparaitre si on a un seul constructeur qui accepte l'implicite, mais bizarrement les problèmes apparaissent pas si on utilise **objet = valeur** mais ils apparaissent quand on fait **objet = {valeur}.**
- L'appel avec **objet = Objet {valeur}** n'est pas un appel explicite !

Ordre d'initialisation dans le constructeur (chapitre 64)

- L'ordre des attributs dans la classe est important, il faut y faire attention car deux classes qui s'utilisent l'une et l'autre peuvent avoir des attributs non initialisés en fonction de cet ordre. Et si on fait appel a des attributs non initialisé, il y'aura un problème
- Pour + d'info voir le code de NVS, ça ne semble pas très important

Liste d'initialisation et constructeur (chapitre 65)

- Pareil pour ce chapitre, peu de choses à dire si ce n'est que l'ordre dans la liste d'initialisation du constructeur déterminera également l'ordre dans lequel les exceptions seront lancées si il y'en a.
- Pour les exception il existe le **throw std::invalid_argument** et les blocs **try catch** pour les attraper. A noter qu'on peut, comme en Java, écrire notre propre message pour la levée de cette expression, mais on doit le faire en créant un objet std::string en paramètre sinon l'utilisation de l'opérateur + sera impossible pour la concaténation.

Constructeur et attributs constant (chapitre 66)

- Un constructeur qui doit initialiser des attributs constant ne pourra le faire que dans la liste d'initialisation et pas dans le corps du constructeur, sinon erreur de compilation.
- Voir le code de nvs pour creuser, mais pas grand-chose a dire…

Constructeur d'objet avec objets comme attribut (chapitre 67)

- Toujours aussi peu de choses à dire, code à voir soi-même. On peut utiliser des objets comme attributs dans d'autres objets. Si ces attributs ont des constructeurs par défaut ou non, il faudra bien le prévoir dans le liste d'initialisation pour que tout soit initialisé dans chaque attribut, quitte à donner une valeur par défaut dans les constructeur de l'objet qui contiens.

Héritage, constructeur (chapitre 68)

- Pour faire hériter une classe on met dans la définition de la classe cette entête **class Fille : public/private/protected Mere** ceci fera hériter Fille de Mere.
- Il faut savoir que quand on crée un objet de la fille, on fera appel au constructeur de la mère pour créer la partie « Mere » de l'objet Fille. Ainsi si on fait appel au constructeur de la fille, elle fera appel au constructeur par défaut de la mère pour créer la partie mère de l'objet
- Cet appel du constructeur de la mère est fait avant un probable appel du constructeur de la fille.

- La fille peut faire appel à un autre constructeur que celui par défaut de la mère. On le fait lors de la définition du constructeur de la fille, une fois qu'on est dans la liste d'initialisation de la fille on fait « **Mere {valeur}** » entre les différentes valeurs des attributs.
- On ne peut pas utiliser de constructeur par défaut pour la fille si la mère n'en a pas un, on aura une erreur de compilation car impossible de créer la partie Mere de la fille.
- Si on utilise le **using Mere::Mere**, on aura pas besoin de créer des constructeurs qui font juste office de relai pour la Mere, ça se fera tout seul et du coup moins d'écriture.
- Parlons rapidement des destructeurs, ce sont des méthodes comme les constructeurs sauf qu'ils détruisent les objets une fois qu'on en a plus besoin, par défaut le destructeur par défaut est très bien, mais on pourrait le redéfinir. On le marque avec un ~ devant le nom de la classe et on le retrouve souvent accompagné du mot clé **virtual** puis un = default si il est déclaré.

Héritage public (chapitre 69)

- Comme dis au-dessus, il existe également un héritage public et un héritage **protected**, chacun a ses spécificités d'accès et on va détailler celles de l'héritage public.
- Dans l'héritage public les accès sont donc à détailler. Lorsqu'une fille hérite d'une mère, on pourra accéder dans la classe à tous les attributs de la fille + les attributs publics et protected de la mère. Mais si on essaye d'accéder aux attributs privés de la mère, erreur de compilation.
- Pour l'accès aux attributs d'une autre fille c'est la même chose, on a accès dans la classe à tous les membres de la fille, mais pas les membres privés de la mère.
- Pour l'accès d'une mère qui sera dans la classe fille (et donc pas sa classe), elle n'aura pas accès à ses attributs privés et protected. Pas de blem pour attributs public. Elle n'aura pas accès aux attributs des objet fille car pour eux ils n'existent pas.
- En dehors des classes, les mères et les filles ont accès à leurs attributs public respectives, et la fille a accès aux attributs public de sa mère, pas l'inverse.

- Lors de la destruction de la Fille, si elle n'a pas de destructeur, on utilise celui de la mère, mais on détruira une partie de l'objet, pas tout, du coup problème de fuite de mémoire, le mieux c'est de laisser le destructeur automatique et utiliser un destructeur virtuel pour faire ça.

Héritage final (chapitre 70)

- Le mot clé final est contextuel, c'est-à-dire qu'on pourra s'en servir pour une variable mais qu'accompagné d'une classe il devient un mot clé.
- On le met dans la définition de la classe, juste après le nom de la classe. Si on le met, on ne pourra plus hériter de cette classe et si on tente de le faire, erreur de compilation.
- On peut utiliser le mot clé final sur une méthode uniquement si elle est virtuelle, voit chapitre 75.

Héritage public/protected/private (chapitre 71)

- L'héritage privé est l'héritage utilisé par défaut. C'est-à-dire que si on fait **class Fille : Mere**, c'est par défaut de l'héritage privé.
- Pour les détails sur l'héritage public, voir le chapitre 69 mais pour rappel. La mère aura accès à ses attribut publique et la fille a accès aux attributs publiques de sa mère et d'elle-même. Dans la classe, la fille aura accès à tous ses attributs ainsi qu'aux attributs protected et private de la mère ainsi qu'à ses méthodes publiques.
- Parlons maintenant de l'héritage **protected**. La fille aura accès en dehors de la classe juste à ses attributs public, alors que dans la classe, la fille aura accès a tous ses attributs ainsi que les attributs protected et private de la mère, mais cette fois ci, pas d'accès aux méthodes public de la mère.
- Pour l'héritage **private,** c'est exactement la même chose que l'héritage protected, les différences apparaissent a la petite descendance de la Mere.
- Dans la petite descendance de Mere avec l'héritage public (et on fait descendre encore avec un héritage public). On a accès aux méthodes public de la grand-Mere et de la Mere, ainsi que leurs attributs public et protected. Ceci vaut dans la classe, en dehors de la classe, on a accès aux attributs public de la grand-mère et de la mere.

- Dans la petite descendance de Mere avec l'héritage protected, on a pas accès aux attributs et aux méthodes de la grand-mère ainsi qu'aux attributs de la mere en dehors de la classe, dans la classe on a accès aux attributs protected et private a la grand-mère et a la mere via la mere. On a également accès aux méthodes public de la Mere.
- Dans la petite descendance de l'héritage private, on a accès que aux attributs public de la petite fille, pas aux attributs public de la mère ou la grand-mère, comme en héritage protected. Dans la classe, on a plus accès aux attributs de la grand-mère, on a accès aux attributs protected et public de la mère, c'est tout.
- Toutes les petites descendances décrites au-dessus ont eu un héritage public de la Fille a la petite fille, et les différences apparaissent.
- Pour la construction par défaut d'un objet, l'héritage private ne pose pas de problème car la petite fille appelle le constructeur par défaut de la mere et même si celui de la grand-mère n'est pas accessible y'a pas de problème.

Héritage et masquage de portée (chapitre 72)

- Le **masquage de portée** permet d'avoir accès qu'aux méthodes ou attributs d'une mère, ce qui peut être utilisé quand on a des surcharges d'une même fonction pour la mère et la fille.
- Ainsi si la mère hérite de la fille, dans le main, on pourra choisir d'avoir accès aux méthodes de la mère en faisant **objetFille.Mere::fonction()** ainsi on appellera les fonction de la mère, par contre dans ce cas, on ne prend pas en compte les fonction surchargée si il y'en a chez la fille.
- Pour utiliser les fonction de la mère et de la fille, on met un **using Mere::fonction** dans la classe fille, ainsi la fonction quand on l'appellera prendra en compte les fonctions de la mère et de la fille.
- Par contre, on ne pourra pas faire appel à la méthode de la Mere dans la méthode de la Fille, si on tente de le faire, erreur a l'exécution.
- Parlons rapidement des **struct**. Ce sont comme des classes, dans le sens où elles peuvent hériter, avoir des attributs et des méthodes, mais elles ne respectent pas l'encapsulation, tout est **public** par défaut dans une struct, même l'héritage si on ne met rien sera par défaut **public**

Héritage et polymorphisme (chapitre 73)

- Le polymorphisme dont on traitera ici est le polymorphisme dit « vertical » car il est lié a l'héritage. Le polymorphisme en lui-même est défini par le fait que les objets qui héritent d'une Mere peuvent prendre la forme fille mais également la forme mère de l'objet.
- Mais pour pouvoir utiliser cette fonctionnalité de plusieurs formes prises, il faut utiliser le mot clé **virtual**, grâce à ça, la méthode dite virtual pourra être utilisée en prenant une autre forme.
- Le mot clé **virtual** pour les méthodes ne doit être utilisé que dans la classe, si on le fait en dehors, erreur de compilation.
- Si on crée un objet mère et qu'on utilise une fonction virtuelle ou pas, on utilisera en tout cas toujours la forme de la mère. Pareil pour un objet Fille, méthode virtual ou pas, on utilisera toujours la forme de la fille pour faire appel a ces méthodes.
- Par contre, on pourra faire mere = fille, vu qu'une partie de la fille est en fait l'objet Mere. Si on essaye d'utiliser les méthodes, c'est toujours les méthodes de la mère qui seront utilisées, si on essaye d'utiliser les méthodes de la fille, erreur de compilation car l'objet crée a la base est un objet Mere.
- Voyons ça avec les références maintenant. Si on crée une référence de Mere et qu'on la lie a un objet mere. En appelant les méthodes de la Mere, on aurait les mêmes résultats, c'est-à-dire que c'est les méthodes mères qui sont appelés.
- Si on force le cast d'une référence de Mere vers une référence de Mere via le **dynamic_cast** qui fait des vérifications (oui, du même type vers le même type), ça marchera sans soucis. Par contre si on essaye de faire un **dynamic_cast** vers une référence de Fille, l'exception sera levée.
- Si on tente de faire un dynamic cast avec un objet avec lequel on a juste la mere ou rien en commun, erreur de compilation
- Si on créer une référence de Mere mais qu'on lie ça a une Fille, alors quand on utilise une méthode virtuelle on appelle celle de la Fille, si la méthode est non virtuelle on appelle celle de la Mere.
- Si on tente le dynamic_cast vers la Fille ou vers la Mere ça fonctionne sans soucis, et ça appelle les méthodes de la Mere ou la Fille sans soucis !

- Si on utilise les pointeur et qu'on pointe vers une référence de Mere. On appellera toujours les méthodes de Mere, virtual ou pas. Et le dynamic_cast ne fontionnera que vers un pointeur de référence de Mere.
- Si on fait pointer un pointeur de Mere vers une référence de Fille, la méthode non virtuelle appellera la méthode de Mere, la méthode virtuelle celle de Fille, et le dynamic_cast fonctionnera pour Mere et Fille.
- Si on utilise le mot clé **override** (qui est un mot clé contextuel comme final), pour qu'une méthode puisse être déclarée override, elle doit avoir exactement la même signature (cad le même nom, parametre et type de retour) que dans la classe Mere, il sers surtout a la vérification de la signature d'une méthode, mais aussi a forcer l'utilisation d'une méthode.
- Si on essaye d'utiliser les méthodes d'un fille même si ils sont virtual, on aura jamais le choix avec la méthode de la Mere, aucune ambiguité possible si on ne procède pas avec les pointeur ou les référence, aucun moyen de faire du polymorphisme.

Classe abstraite (chapitre 74)

- Une classe abstraite est une classe dont on ne pourra pas créer d'objet, elle est là juste pour qu'on hérite de ses méthodes. Pour déclarer une classe abstract, il faut déclarer une méthode virtual dans la classe et mettre = 0 ; ainsi on ne pourra pas créer d'objet. Par contre on pourra créer une référence ou un pointeur vers un objet Fille.
- Contrairement au polymorphisme, même si cette méthode est virtual, on utilisera la méthode de la fille car la méthode de la Mere est = 0 ou est définie.
- On peut faire hériter une classe abstraite ou la faire hériter.

Virtual et Final (chapitre 75)

- Seules les méthodes virtuelles peuvent être final, l'utilisation du mot clé ici signifie que cette méthode ne pourra pas être réécrite dans la classe Fille. Dans ce cas là l'appel des méthodes d'un objet Fille appellera la méthode de la Mere. Malgré tout, je ne sais pas pourquoi parfois ça compile et accepte les surcharges alors qu'il y'a final… ça devrais bug

- Si le mot clé **override** est présent chez une méthode de la Fille, c'est toujours celle-ci qui sera utilisée si on pointe ou on utilise une Fille. Le mot clé override ne servira pas dans la classe Mere, a moins qu'elle-même n'hérite.

---------- FIN ICI : J'étais pas là pour les 5 derniers chapitre