# ❖ Programmation Avancée 1 ❖

## Chapitre 1 - Concepts du C++

**Table des matières**

# GESTION DE LA MÉMOIRE

- La mémoire dans un ordinateur est une succession d'octets (8 bits), organisés les uns à la suite des autres, accessibles par une adresse.
- En C++, la mémoire pour stocker des variables est organisée en deux catégories, c'est-à-dire la pile (**stack**) et le tas (**heap**).

# La pile

- La pile est un espace mémoire réservé au stockage des variables désallouées automatiquement.
- La pile utilise le concept de LIFO (Last In First Out).
- Voyons la pile comme une pile d'assiettes, où on a le droit d'empiler/dépiler qu'une seule assiette à la fois.
- Par contre, on peut empiler des assiettes de différentes grosseurs.
- Lorsqu'une assiette est dépilée, l'assiette est détruite.

# Le tas

- Le tas est l'autre segment de mémoire utilisé lors de l'allocation dynamique de mémoire.
- Sa taille est souvent considérée comme 'illimitée', malgré le fait qu'elle est limitée.
- Les fonctions utilisées pour attribuer de la mémoire en C (malloc, free) et en C++ (new, delete) permettent respectivement d'allouer et de désallouer de la mémoire sur le tas.
- TOUTE mémoire allouée dans le tas doit être désallouée explicitement!

- L'allocation d'une nouvelle zone mémoire se fait sur le tas.
- Elle se fait en utilisant la fonction malloc. Le prototype de cette fonction est la suivante:

```
void *malloc(size_t taille);
```

- L'argument donné en paramètre est la taille en octets de la zone mémoire désirée.
- La valeur retournée est un pointeur **void*** sur la zone mémoire allouée, ou **NULL** en cas d'échec.
- Vous pouvez redimensionner un espace mémoire déjà alloué dynamiquement avec la fonction realloc.

- La mémoire allouée dynamiquement, doit, à un moment ou un autre, être libérée.
- Cette libération de mémoire se fait par la fonction free. Voici la signature de cette fonction:

```
void free(void *pointeur);
```

- Important:
  - La mémoire doit être libérée une seule fois;
  - Impossible d'accéder à nouveau à l'espace mémoire une fois désallouée;

Example MALLOC et FREE

- Voici quelques règles qui permettent de vous sauver quelques troubles lors d'allocation de mémoire dynamiquement:
  - Toute déclaration de pointeur s'accompagne de son initialisation à NULL.
  - Avant tout appel de malloc, on doit s'assurer que le pointeur à allouer est bien NULL.
  - Après tout appel de malloc, on s'assure qu'aucune erreur ne s'est produite.
  - Avant de libérer une zone mémoire, on s'assure que son pointeur n'est pas NULL.
  - Dès qu'une zone mémoire vient d'être libérée par free, on réinitialise son pointeur à NULL.

- Pour allouer dynamiquement des espaces mémoire en C++, les opérateurs new et delete seront utilisés.
- L'opérateur new renvoie une adresse où est créée la variable.
- Il faudra donc un pointeur pour conserver le lien vers cette variable.
- Manipuler ce pointeur reviendra à manipuler la variable allouée dynamiquement.
- Pour libérer cette espace mémoire, l'opérateur delete suivi du pointeur sera utilisé.

- Pour allouer dynamiquement des espaces mémoire en C++, les opérateurs new et delete seront utilisés.
- L'opérateur new renvoie une adresse où est créée la variable.
- Il faudra donc un pointeur pour conserver le lien vers cette variable.
- Manipuler ce pointeur reviendra à manipuler la variable allouée dynamiquement.
- Pour libérer cette espace mémoire, l'opérateur delete suivi du pointeur sera utilisé.

Example NEW et DELETE

# Fuite de mémoire

- Les fuites de mémoire (memory leaks) surviennent lorsque la désallocation d'un pointeur allouée dynamiquement n'est pas faite, et que le programme se termine.
- Aucun programme ne possède désormais de lien vers l'adresse mémoire, et ce bloc de mémoire reste en mémoire indéfiniment, jusqu'à redémarrage de l'ordinateur.
- Chaque allocation avec **new** doit impérativement être détruite avec **delete**.

- La fuite de mémoire est une zone mémoire qui a été allouée dans le tas par un programme qui a omis de la désallouer avant de se terminer.
- Cela rend la zone inaccessible à toute application (y compris l'OS).
- Si ce phénomène se produit trop fréquemment, la mémoire se remplit de fuite et le système finit par planter, faute de mémoire.
- Certains langages (ex: Java, C#, etc.) ont introduit un système de 'ramasser-miettes' (Garbage Collector).
- Ce système permet de gérer ces allocations de mémoire pour éviter d'obtenir des fuites de mémoire.

# CONCEPTS DE BASE

# Constructeur / Destructeur

- Le constructeur est la fonction membre d'une classe ou struct qui est appelée automatiquement lors de la création de l'objet.
- Comme cette fonction est appelée en premier, elle est très utile pour l'initialisation de variables.
- Le constructeur d'un object porte le même nom que la classe, et ne possède aucune valeur de retour (même pas un **void**).
- Un constructeur peut avoir des paramètres de construction.
- La définition d'un constructeur est optionnelle. S'il n'est pas défini, le **compilateur** va en créer un lors de la compilation, vide.

Exemple Constructeur

- L'ordre de construction est la suivante:
  1. Il appelle les constructeurs membres et de classe de base dans l'ordre de déclaration.
  2. Si la classe est dérivée de classes de base virtuelles, il initialise les pointeurs de base virtuels de l'objet.
  3. Si la classe possède ou hérite des fonctions virtuelles, il initialise les pointeurs de fonction virtuelle de l'objet. Les pointeurs de fonction virtuelle pointent sur la table de fonctions virtuelles de la classe pour permettre la liaison correcte des appels de fonction virtuelle au code.
  4. Il exécute le code dans son corps de fonction.

Exemple Ordre Constructeurs

- Initialisez les membres de classe à partir des arguments de constructeur à l'aide d'une **liste d'initialiseurs membres**.
- Cette méthode utilise l'initialisation *directe,* qui est plus efficace que l'utilisateur d'opérateurs d'assignation dans le corps du constructeur.

Chapitre1_MemberInitList

- Si une classe a un constructeur avec un paramètre unique, ou si tous les paramètres sauf un ont une valeur par défaut, le type de paramètre peut être implicitement converti en type de classe.

Exemple Explicit Constructor

- Parfois, ces conversions peuvent être utiles. Cependant, la plupart du temps, elles peuvent entraîner des erreurs subtiles mais graves.
- En règle générale, vous devez utiliser le mot clé **explicit** sur un constructeur afin d'éviter que le compilateur puisse faire la conversion de type.

- Si le constructeur est explicite, l'utilisation du constructeur en passant seulement un **int** lorsque le paramètre est de type **Box**, car le constructeur de cet objet prend en paramètre un **int**, générera une erreur de compilateur.

- Il existe un type de constructeur qui est le constructeur par défaut.
- Les constructeurs par défauts n'ont aucun paramètre.
- Voici quelques règles des constructeurs par défaut:
  1. Les constructeurs par défaut sont déclarés par le compilateur si aucun constructeur n'a été déclaré.
  2. Un constructeur par défaut n'a pas besoin de parenthèse lors de la construction de l'objet.
  3. Si une classe n'a aucun constructeur par défaut, il est impossible de déclarer un tableau d'objets de cette classe à l'aide de la syntaxe avec crochets.

# Les fonctions inline

- En C++, il a un nouveau mot-clé appelé inline.
- Lorsque le compilateur voit ce mot, il se charge de remplacer le code où l'appel de cette fonction est fait par le code directement.

Exemple Fonction Inline

- On a pu remarqué le mot-clé inline au début du prototype ET au début de la définition de la fonction.
- Cela signifie, pour le compilateur: “À chaque fois qu’on fera appel à la fonction carre, je placerai directement le code de cette fonction à l’endroit de l’appel.”
- La version compile du code précédemment dans l’exemple serait:

```
1  int main()
2  {
3      cout << 10 * 10 << endl;
4
5      return 0;
6  }
7
```

- L'avantage est que l'exécution du programme sera plus rapide, surtout si la fonction est appelée plusieurs fois.
- Lors d'un appel ''classique" de fonction, le processeur va sauter à l'adresse de la fonction en mémoire, retenir l'adresse où il doit revenir après la fonction.
- Ce processus est assez rapide. Cependant, si la fonction est appelée très souvent, il est préférable de l'**inliner** pour éviter toutes ces étapes.

- Le désavantage est que si vos fonctions sont longues, votre programme va grossir en termes d'exécutable.
- En effet, étant donné que la fonction est copiée telle quelle à l'endroit de l'appel, toutes les lignes de la fonction sont dupliquées.
- En règle général, les fonctions inline sont des fonctions très courtes (voir 1 ligne) que l'on est susceptible de réutiliser souvent (ex: **getter** / **setter**).

# La surcharge d'opérateurs

- Une autre des fonctionnalités très pratique est la surcharge des opérateurs.
- Permets de réaliser des opérations mathématiques intelligentes entre les différents objets que vous aurez créés.
- Applicable sur les symboles comme +, -, *, ==, <, etc.

- Disons que nous avons des objets qui stock des durées de temps, nous pouvons par exemple, comparer si deux objets de type Duree est identiques, si un est plus bas qu'un autre, addition deux objets du même type, etc.
- Le problème, c'est que c'est des objets complexes, et que ces types d'opérations en s'applique pas sur ces types.

- Nous pourrions utiliser la surcharge des opérateurs pour définir ces opérateurs et implémenter ce que ces opérations doivent être.
- Bien sûr, la surcharge d'opérateur ne s'applique pas à toute les classes.
- Par exemple, nous avons une classe Personnage, surchargé l'opérateur d'addition ne ferait aucun sens.
- Voyons comment nous pouvons surcharger ces opérateurs

Exemple Surcharge Operateurs

# Le mot-clé friend

- Dans les langages orientés objet, l'amitié est le fait de donner un accès complet aux éléments d'une classe.
- ATTENTION! En déclarant une fonction friend d'une classe, on casse complètement l'**encapsulation** de la classe, puisque quelque chose d'extérieur à la classe pourra modifier ce qu'elle contient.
- Il faut faire très attention quand nous utilisons le mot-clé friend, surtout que l'**encapsulation** est l'élément le plus important en orienté-objet.
- Mais pourquoi utiliser le mot-clé friend?
- Bizarrement, le mot-clé friend peut renforcer l'encapsulation.

- Revenons avec notre classe Duree faite précédemment.
- Nous avons surchargé l'opérateur << pour pouvoir afficher une durée, directement avec l'opérateur de flux de sortie.
- Nous avons dû créer une fonction public afficher(). Donc, la fonction donne un service supplémentaire à l'utilisateur. Cependant, la fonction afficher n'est utilisé que par <<, et même si un utilisateur l'utilisait, cela n'impactera pas l'intégrité de l'objet.
- Pourquoi ne pas mettre la fonction affichée en privé?

- Les utilisateurs n'auraient plus accès à la fonction, mais l'opérateur << non plus.
- C'est à ce moment que le mot-clé friend fait son entrée!
- Si l'opérateur << est déclaré comme friend de la classe Duree, alors il aura accès à la partie privée de la classe.
- Pour déclarer une fonction friend:

```
1  friend std::ostream& operator<< (std::ostream& flux, Duree const& duree);
```

Exemple Friend

- S'assurer que ce qui utilise le mot-clé friend ne vient pas détruire la classe ou modifier ses attributs.
- Si vous avez besoin d'une telle méthode, faites plutôt une fonction membre de la classe.
- Voici deux règles qui doivent être respectées pour utiliser le mot-clé friend:
  1. Une fonction friend ne doit pas, en principe, modifier l'instance de la classe.
  2. Les fonctions amies ne doivent être utilisées que si vous ne pouvez pas faire autrement.

# Pointeurs

- Les pointeurs sont une fonctionnalité de base du langage C.
- Comme le C++ est bâti sur le C, il en hérite par ce fait.
- Précédemment, nous avons vu comment jouer avec la mémoire d'un ordinateur.
- On se rappelle que la mémoire est en fait une longue série de cases (ou octets (8bits)), et que ses cases sont numérotées avec des adresses.
- Lorsque nous créons une variable, la valeur est mise dans une case mémoire.

- Quelques questions peuvent se poser:
  - Comment ces variables sont-elles disposées en mémoire?
  - Sont-elles placées les unes à la suite des autres ou éparpillées dans la mémoire?
- En regardant les adresses mémoires, on pourrait savoir comment elles sont placées en mémoire.
- Pour savoir la taille d'une variable en mémoire, la fonction sizeof peut être utilisée. (ex: sizeof(int) va être égal à 4 octets)
- L'opérateur & suivit d'une variable permet d'obtenir l'adresse mémoire de la variable.

Exemple & et sizeof

- Mais que signifient ces adresses? Il s'agit de nombres écrits en base 16 (hexadécimal).
- En C/C++, un nombre hexadécimal commence toujours par **0x**. Cela évite les confusions avec les autres types de nombres.
- Ce qui suit le **0x** est simplement la valeur du nombre.
- Un hexadécimal est codé sur 4 bits. Nous pouvons donc dire qu'une case mémoire est de 32 bits (8*4 = 32 bits), soit 4 octets.

- C'est important de bien comprendre qu'une adresse de variable n'a aucun rapport avec sa valeur ou son type.
- L'adresse d'une variable est déterminée par le compilateur.
- Que l'ont ai des **int**, **double**, **char**, **string** ou une **classe custom**, les adresses seront toujours codées sur 32 bits (ou 64).
- Si la classe a besoin de 64 bits en mémoire, alors la classe remplira deux cases mémoire (ou une dans le cas d'une architecture 64 bits).

- Un pointeur contient l'adresse d'une variable du type déclaré.
- Lorsque nous déclarons une variable, "**int a**" signifie que **a** est une variable de type **int**.
- Par le fait même, nous  pouvons dirons que dans "**int* p**", **p** est un pointeur sur une variable de type **int**.
- Le pointeur est le plus souvent typé, donc le pointeur ne peut pointer que sur un seul type de variable.

- Il existe plusieurs façons de déclarer un pointeur:

```
int* p;
int *q;
int * r;
```

- Toutes sont correctes.
- Attention cependant. Cette façon de déclarer deux pointeurs

```
int* p, q;
```

n'est pas valide. En fait, cela déclare un pointeur de **int**, ainsi qu'un **int**.

- C'est bien beau toutes ces petites étoiles un peu partout dans le code. Mais ça sert à quoi?
- **L'indirection** est utile pour connaitre la valeur stockée à une adresse contenue dans un pointer.

Exemple_Indirection

- Il existe une valeur spéciale pour les pointeurs, la valeur **0** (ou **NULL**).
- En fait, **NULL** est un 'define' en C qui est défini à **0**.
- La case 0 de la mémoire n'est pas accessible. On considère donc que l'adresse 0 n'existe pas.
- En assignant **0** ou **NULL** à un pointeur, on considère que le pointeur ne contient pas d'adresse.
- C'est souvent un test à effectuer avant d'effectuer des opérations sur un pointeur. (ex: if (p != NULL) { … } ).

- L'importance d'initialiser un pointeur à NULL ou 0 lors de sa création est primordiale.
- Les pointeurs fous sont des pointeurs qui pointent à un endroit où le programme ne peut pas écrire.
- En effet, un programme à son espace en mémoire où il peut écrire. À l'extérieur de cette espace, le programme ne peut pas écrire.
- Si vous déclarez un pointeur, sans valeur initiale, ce pointeur peut pointer dans un endroit où il est interdit d'écrire.
- Si vous tentez d'utiliser ce pointeur pour écrire dans cette espace mémoire, vous risquez au mieux de planter le programme.

- Le passage de l'adresse d'une variable dans une fonction est un concept important.
- Dans un appel de fonction, les paramètres peuvent être passés par copie ou par référence.
- Lors d'un appel de fonction avec paramètres par copie, la valeur de ceux-ci est recopiée dans un autre variable, utilisé à l'intérieur de la fonction.
- Cependant, avec les pointeurs, nous pouvons passer par référence une variable, ce qui évite de recopier en mémoire la variable.

Exemple Pointeur Parametres

- Le principe de la fonction **Swap** sans pointeur est bon. La déclaration d'une variable temporaire pour stocker la valeur de a, faire la rotation de b vers a et réassigner la valeur de la variable temporaire à b permet de faire un 'swap' de valeurs.
- Cependant, a et b dans cette fonction sont des copies locales!
- Même si on les affecte à une autre valeur, cela n'a aucun effet sur les variables dans le main.

- C'est là que les pointeurs deviennent utiles!

- La fonction Swap avec pointeurs attend maintenant deux pointeurs vers des int.
- On donne donc l'adresse respective de x et de y, et c'est adresse sont copiées dans a et b.
- À ce moment-là, en modifiant les valeurs qui se trouvent aux adresses contenues dans a et b, on modifie directement les variables dont on a donné l'adresse lors de l'appel, c'est-à-dire x et y.

- Bien sûr, ce n'est pas forcément pour modifier une valeur que nous passons des paramètres par référence.
- Prenons, par exemple, une **classe** qui contient beaucoup de variables membres.
- Faire une copie de cette **classe** à chaque appel de la fonction serait du gaspillage de ressource.
- Dans ce cas-là, passez tout simplement son adresse et le tour est joué!
- En règle générale, il vaut mieux passer une **struct** ou une **classe** par adresse.

- Il existe une relation en C/C++ entre les **tableaux** et les pointeurs.
- Nous pouvons obtenir, comme sur n'importe variable, l'adresse mémoire reliée, avec l'opérateur &.

```
double tabl[10];
cout << "adresse de tabl[0] = " << &tabl[0] << endl;
```

- Il n'existe pas, en C++, de type tableau. Pourtant, nous en avons déclaré un…!?
- En vrai, **tabl** est un pointeur vers un double, qui se trouve le premier élément d'un tableau de doubles.

- Revenons sur la notation entre crochets. Que signifie tabl[5]?
- Effectivement, cela signifie le sixième élément du tableau. C'est tout ce qu'il nous faut, rien de plus.
- Mais côté compilateur, c'est autre chose!
- Il connaît la taille en mémoire de nos objets (ici, des doubles). En supposant que chaque objet ''mesure" n octets, tabl[5] est situé à 5*n octets du début de la zone mémoire occupée par le tableau.

- Parlons un peu d'arithmétique sur les pointeurs.
- On peut faire des calculs avec les pointeurs, mais certaines règles s'appliquent.
- Un pointeur contient une adresse, qui exprime un décalage, en octets, depuis l'adresse 0.

Exemple arithmétique pointeurs

- Si nous faisons la différence entre l'adresse suivante et l'adresse de a, nous obtenons un décalage du nombre d'octets occupé par un int.
- Pourtant, nous avons incrémenté le pointeur p que de 1 (++p).
- En réalité, lorsqu'on travailler avec les pointeurs, l'unité ne vaut plus 1, mais la taille de l'objet vers lequel on pointe. Donc ++p incrémente de 1*sizeof(int) = 4 l'adresse contenue dans p.

- Pour en revenir à notre notation de tabl[5], nous pouvons donc en déduire qu'il existe plus d'une méthode pour accéder aux valeurs d'un tableau.
- Voici les deux variantes qui permettent d'accéder aux valeurs:

```
int tabl[10];
tabl[5] = 23;cout << "tabl[5] vaut " << tabl[5] << " ou encore " << *(tabl + 5) << endl;
```

- La première façon est la façon classique.
- La seconde est une façon détournée. *(tabl + 5) signifie *la valeur contenue à l'adresse contenue dans **tabl** plus un décalage de 5 fois le nombre d'octets occupés par un int.*

- Si vous indiquez entre les crochets un décalage dépassant la taille limite du tableau, le compilateur n'y voit que du feu.
- Il va aller mettre un pointeur là où vous lui indiquez, et va faire à cet endroit tout ce que vous lui demandez de faire.. S'il y a le droit d'écrire à cette espace mémoire.

Exemple Copy Array Avec Pointeur

- Examinons maintenant comment utiliser les pointeurs avec une structure (ou une classe).
- La seule différence sera dans la manière d'accéder à une variable membre d'une **struct** dont on a l'adresse.

Exemple Struct Pointers

- Il existe aussi des pointeurs de pointeurs.
- En effet, il est envisageable de faire pointer un pointeur vers un autre pointeur, étant donné qu'un pointeur est une variable.
- Voici comment déclarer un pointeur de pointeur:

```
int a = 14;
int* p = &a;
int** pp = &p;
```

- Mais à quoi cela peut servir?

- Les pointeurs de pointeurs servent à modifier un pointeur dans une fonction où l'on passe en paramètre un pointeur.

Exemple pointer pointer

- Si un pointeur contient une adresse mémoire, comment se fait-il qu'on soit obligé de respecter un type pour le contenu de cette mémoire?
- Il existe un moyen de pointer sur un peu tout et n'importe quoi.
- Vous vous rappelez du type **void**?
- C'est grâce à lui qu'on va préciser que l'on ne donne pas de type particulier à un pointeur.
- Cependant, lorsque vous travaillez avec un pointeur **void**, vous ne pouvez pas faire d'indirection comme d'habitude.

- Il faut effectuer une conversion pour pouvoir récupérer la valeur contenue à l'adresse en question:

```
int a = 14;
void* p = &a;
cout << *((int*)p);
```

- Un pointeur de type **void** peut recevoir l'adresse de n'importe quel type de donnée. Néanmoins, une conversion s'impose lors de l'indirection ou du déréférencement.

Exemple Void Pointer

- Retour sur les notions des pointeurs:
  - L'opérateur & permet de connaître l'adresse en mémoire d'une variable.
  - Un pointeur est le plus souvent typé, c'est-à-dire qu'il ne peut pointer que sur un seul type de variable.
  - L'opérateur d'indirection * permet de connaître la valeur stockée à une adresse contenue dans un pointeur.
  - Il est fortement recommandé d'initialiser tous ses pointeurs lors de leur déclaration, afin d'éviter les pointeurs fous. Aucune variable ne peut être stockée à l'adresse NULL (0), c'est donc une valeur qu'on utilise pour l'initialisation ou pour savoir qu'un pointeur n'est pas valide.

- On peut passer l'adresse d'une variable comme paramètre à une fonction. Ceci est utile lorsqu'on veut changer la valeur de cette variable depuis la fonction ou si on passe une structure ou classe très grande.
- Lorsqu'on déclare un tableau tabl, tabl est un pointeur vers le type déclaré.
- L'opérateur de déréférencement –> permet d'accéder aux membres d'une structure de données dont on connaît l'adresse.
- Un pointeur de type void peut recevoir l'adresse de n'importe qul type de donnée. Néanmoins, une conversion s'impose lors de l'indirection ou du déréférencement.

Exemple Sort Array

# Références

- Le C++ à introduit une nouvelle fonctionnalité, les références!
- Les références ressemblent beaucoup aux pointeurs. Elles ont été créées pour simplifier l'utilisation de ces derniers.
- Une référence est créer lors d'une déclaration de variable avec l'opérateur préfixe **&.**

```
1 int &referenceSurAge;
2
```

- Cependant, si vous tentez de compiler le code dans la page précédente, vous obtiendrez une erreur de compilation:

```
error: 'referenceSurAge' declared as reference but not initialized
```

- Pourquoi?
- L'erreur est assez éloquente.
- Contrairement à un pointeur, le compilateur s'attend à ce qu'une référence doive être initialisée à la déclaration.
- De plus, une fois initialisée, la référence ne pourra plus changer!

- Pour résumer, il y a deux règles que vous devez vous rappeler lorsque vous utilisez une référence:
  1. Une référence doit être initialisée dès sa déclaration.
  2. Une fois initialisée, une référence ne peut plus être modifiée.
- Pour initialiser une référence, vous avez juste besoin d'écrire le nom de la variable dont elle sera le synonyme. Pas besoin d'assigner une adresse comme avec un pointeur:

```
1 int age = 21;
2 int &referenceSurAge = age;
```

- L'utilisation d'une référence se fait de la même manière qu'une variable normale!
- Pas besoin d'ajouter une * devant la variable comme un pointeur.

Exemple Utilisation Reference

- Nous avons vu qu'une référence s'utilise de la même marnière que la variable d'origine.
- C'est le compilateur qui fait la 'conversion' et qui sait qu'il doit faire le lien entre la variable 'age' et la référence.

- Mais pourquoi utiliser les variables, si les pointeurs peuvent faire le même travail?
- Faisons une comparaison entre les deux.
- Les références sont une alternative aux pointeurs, introduits en C++. Les pointeurs sont une fonctionnalité du C.
- Les références ont l'avantage d'être plus simples à utiliser, mais elles ne peuvent pas remplacer à 100% les pointeurs.
- Pourquoi?

- Simple! Une référence ne peut pas faire référence à une nouvelle variable une fois qu'elle a été initialisée.
- Un pointeur, lui, peut toujours pointer vers une nouvelle adresse au cours de l'exécution du programme.

- Dans la plupart des langages de haut-niveau (ex: Java), les pointeurs n'existent plus.

- Il vous sera très facile de confondre l'utilisation des pointeurs et des références dans vos débuts.
- Je vais vous montrer l'utilisation, pour le même code, de références versus les pointeurs.

Exemple Pointeurs VS References

- Ce qu'il faut retenir, c'est que l'utilisation de références simplifie l'écriture du code, en supprimant le symbole * de l'équation. De plus, le risque de pointeur NULL ou ne pointant pas au bon endroit est minimisé.

- Les références peuvent aussi avoir référence sur une structure ou une classe.
- La différence entre les pointeurs (on se rappel, les pointeurs doivent utiliser la –> pour accéder à un objet du pointeur), une référence utilise le '.'
- Encore une fois, on voit que l'utilisation d'une référence se fait comme avec une variable.

Exemple Struct Reference

- Le cas le plus important à comprendre est l'utilisation des références dans les appels de fonctions.
- La véritable utilité d'une référence (ou pointeur) dans un appel de fonction est primordiale lors d'envoyer une structure ou classe vers une fonction.

Exemple Reference Parametre

- On transmet la référence à la fonction **RemiseAZero** sans même à devoir passer l'adresse de la structure **point.**
- La fonction doit préciser qu'elle reçoit une référence. On utilise donc l'opérateur **&** devant la variable en paramètre.
- De plus, cela permet de ne pas copier en mémoire l'objet en paramètre.
- Dans la plupart des cas (si ce n'est pas toujours), l'appel d'une fonction avec des références en paramètres est utilisé.

Exemple Pointer Reference Parametres

# Constantes

- Nous étudierons un principe très important, le **const-correctness**.
- La définition de ce concept est simple: utiliser correctement la **constance** dans un code.
- La constance en C++ est créée avec le mot-clé **const**.
- Ce mot-clé permet de déclarer une variable comme constante, c'est-à-dire que sa valeur n'est pas censée changer au cours de l'exécution du programme.

- La **const-correctness** inclue également l'utilisation d'un mot-clé et d'une opérateur que l'on emploie moins, mais que nous allons étudier: **mutable** et **const_cast**.
- Ses deux opérateurs permettent d'ignorer la constance d'une variable.
- Effectivement, on pourrait se demander pourquoi utiliser le **const**, si nous pouvons les modifier en cours d'exécution avec **mutable** et **const_cast**?

- Hormis ce cas, la const-correctness est utile au programmeur comme sécurité lorsque le code évolue, puisque modifier une valeur d'une constante engendra une erreur de compilation.
- De plus, cela ajoute de la documentation lors de l'utilisation des fonctions et/ou variables.

- Le mot-clé **const** empêche toute modification ultérieure de la valeur d'une variable.
- Toute tentative de changement d'état sur la variable sera détectée comme une erreur à la compilation.
- Dans le cas qu'une classe ou qu'une structure soit considérée comme constante, tous les sous-objets de l'objet constant seront aussi constants.
- Voici comment déclarer une variable constante:

```
1 int const maVariable;
2
```

- Certains vous diront que la déclaration d'une constante peut aussi se faire de cette manière:

```
1 const int maVariable;
2
```

- Ce n'est pas tout à fait vrai. La règle générale, comme les pointeurs, est que le mot-clé s'applique sur l'élément qui se retrouve directement à sa gauche, ou s'il n'y a rien, à sa droite.

```
1 int i, j;
2 int const * p = &i;
3
4 *p = j; //Erreur, la valeur pointée par p est constante.
5 p = &j; //Correct, le pointeur p n'est pas constant
```

- Alors, comment déclarer un pointeur constant?

```
1  int * const p; //Pointeur constant sur un int non constant
2  const int * const q; //Pointeur constant sur un int constant
3
```

- Un pointeur ne peut voir l'adresse qu'il contient modifiée. À la syntaxe près, il est donc identique à une référence et l'on préfèrera utilisé le dernier.
- Et pour une référence? Comment déclarer une référence constante?

- Faut se rappeler qu'une référence est assignée à une adresse à son initialisation, et ne peut être affectée à une autre adresse par la suite.
- Alors, nous pouvons dire que référence est toujours constante.

- Les fonctions membres peuvent aussi être constantes!
- Le principe d'une fonction constante est qu'elle ne modifie pas l'objet sur lequel elle est appelée.

```
1  struct Exemple
2  {
3      void bar() const
4      {
5      }
6
7  //Ou, si l'implémentation est dissociée de la déclaration :
8      void foo() const;
9  };
10
11 void Exemple::foo() const
12 {
13 }
14
```

- Comme pour les valeurs constantes, le mot-clé vient se placer à droite de ce qui est qualifié mais avant le corps de la fonction.
- Il faut répéter le mot-clé lorsque l'implémentation est séparée de la définition: elle est obligatoire.
- Pour une fonction virtuelle pure:

```
1  struct Exemple
2  {
3      virtual void foo() const = 0;
4  };
5
```

- Cela permet de s'assurer qu'aucune instruction exécutée dans le corps de votre fonction n'affectera aucunement les attributs de votre objet constant.
- Tout ce qui sera utilisé dans votre fonction constante sera des objets constants.

```
1  struct Exemple
2  {
3      void foo() const
4      {
5          mon_int = 0; //Erreur, mon_int est constant.
6      }
7
8      int mon_int;
9  };
10
```

- Cependant, il faut savoir que les paramètres passés dans une fonction constante ne sont pas constants:

```
1  struct Exemple
2  {
3      void foo(Exemple& exemple) const
4      {
5          exemple.mon_int = 0; //Ok, mon_int appartenant à exemple n'est pas
   constant.
6      }
7
8      int mon_int;
9  };
10
```

- Côté optimisation, imaginez-vous un objet très lourd en mémoire, c'est-à-dire un objet qui contient beaucoup de valeurs, variables, fonction, etc. et que vous devez passer cet objet en paramètre constant d'une fonction.
- Nous pourrions penser de le faire de cette manière:

```
1 void maFonction(UneClasseAvecBeaucoupDeDonnees unObjetTresLourd)
2 {
3     //Utilisation d' unObjetTresLourd
4 }
5
```

- Dans l'exemple, l'objet est copié en mémoire, et on se retrouve avec deux instances de notre classe même si nous avions besoin qu'un seul objet.
- De plus, si à l'intérieur de cette fonction, l'objet sera passé à une autre fonction, alors l'objet serait encore copié en mémoire pour rien.
- Au final, nous nous retrouvons avec 3 objets identiques en mémoire, lorsqu'un seul aurait suffi!
- Alors, comment faire?

- Rappelez-vous des références!
- Remplacer le paramètre par une référence règle notre problème de copies en mémoire non utiles:

```
1  void maFonction(const UneClasseAvecBeaucoupDeDonnees& unObjetTresLourd);
2
```

- Au lieu de copier l'intégralité de l'objet, on se contente dans créer une référence qui pointe sur l'objet en question.
- Comme notre objet ne doit pas être modifié par la fonction, le mot-clé const est rajouté.
- De façon générale, on utilisera le passage par référence constante plutôt que par copie.

- Mais qu'arrive-t-il lorsqu'un paramètre d'une fonction est constant, et que nous voulons appeler une fonction non-constante sur le paramètre?
- Essayons-le!

- Le mot-clé **const** à la fin du prototype nous indique que la fonction ne modifie pas l'objet qui l'appelle.
- Lorsque le compilateur voit qu'un objet **const** se fait appeler une fonction qui n'est pas **const**, il ne peut pas garantir que l'objet **const** ne sera pas modifié, d'où l'erreur de compilateur.
- Or, en mettant le prototype **const**, nous disons au compilateur que cette fonction ne peut pas modifier l'objet.

- La compréhension du const est très importante, et son utilisation l'est encore plus!
- Transformer du code non-const en **const-correctness** est une tâche ardue. Ajouter un simple const à une fonction brise tout ce qui appelle cette fonction.
- Règle générale: utiliser le mot-clé const le plus souvent possible!

# Casting

- Le casting est simple. C'est l'action de convertir une valeur d'un type dans un autre type.
- Bien sûr, il faut que le type de départ soit compatible avec le type ciblé.
- Par exemple, un int et un short sont compatibles. Pourquoi?
- Ils ne diffèrent que par leur capacité. En effet, le short peut contenir jusqu'à 32767 tandis que le int va jusqu'à 2147483647.

- Il arrive très souvent que nous devions faire la conversion d'un type vers un autre, et une chance que le C++ nous offre des fonctions pour le faire.
- Il faut cependant être rigoureux dans l'utilisation du casting et de prendre le bon **cast** en fonction du besoin.
- Voici les quatre types de conversions possibles et réalisables en C++:
  1. La conversion **statique** de type
  2. La totale **réinterprétation** des données
  3. La 'conversion' d'un pointeur (ou référence) **constant(e)** vers un pointeur (ou référence) non-constant(e)
  4. La conversion de types **dynamique**

- Une conversion de types dynamique est une conversion qui va s'effectuer pendant l'exécution du programme et non pas le compilateur pour le casting des trois autres types de cast.
- Lorsque le casting dynamique est effectuée pendant l'exécution du code, on peut tomber sur une erreur d'implémentation.
- Si cette erreur se produit, une exception est lancée: std::bad_cast.

- Les mots-clés du C++ pour le casting sont les suivants:
  1. static_cast<>
  2. const_cast<>
  3. dynamic_cast<>
  4. reinterpret_cast<>

# static_cast<>

- Le static_cast<> permet plusieurs choses:
  - Expliciter les conversions *implicites*, supprimant du même fait tout les avertissements que donnerait le compilateur si la conversion peut entraîner un risque (ex: Perte de données d'un **double** vert un **int**).
  - Convertir vers et depuis n'importe quel type pointé à partir d'un **void***. (ex: **void*** vers un **unsigned char***).
  - Convertir au travers d'une hiérarchie de classe, sans effectuer de vérification préalable. (ex: **Base*** vers **Derived*** ou **Base&** vers **Derived&**).
  - Ajouter l'attribut constant au type converti. (ex: **char*** vers **const char***).

- Faut prendre en compte que le static_cast<> ne donnera jamais d'erreur, même si la conversion à échoué.

```
Base* base = new Derived1();

Derived1* derived1 = static_cast<Derived1*>(base);
Derived2* derived2 = static_cast<Derived2*>(base);
```

- Ce code à un comportement indéfini.
- La notion de comportement indéfini n'offre par définition aucune garantie: le code peut avoir le comportement espéré ou faire crasher le programme.

- Le static_cast<> ne permet pas de:
  1. Convertir vers ou depuis un type pointé à partir d'un autre type pointé **autre que void*** (ex: **unsigned char*** vers **char***).
  2. Tester qu'une instance est celle d'un type dérivé (ex: tester qu'un **Base*** est en fait un **Derived***.)
  3. Supprimer l'attribut **constant** du type converti (ex: **const char*** vers **char***).

- En conclusion, static_cast<> est sans doute l'opérateur de conversion que vous serez amené à utiliser le plus.
- Il permet de réaliser des conversions sûres et à pour rôle principal celui d'expliciter les conversions implicites.
- Dans le cas du polymorphisme (oh, les gros mots!), il est préféré à dynamic_cast<> lorsque l'on a **la garantie** que la conversion va réussir.

Exemple static_cast<>

# dynamic_cast<>

- Le seul rôle du dynamic_cast<> est de tester à l'exécution si un pointeur d'un type de base est en fait un pointeur vers un type dérivé.

Exemple dynamic_cast<>

- Pour que dynamic_cast<> fonctionne, le type de base doit posséder au moins une méthode virtuelle.
- Un appel à dynamic_cast<> est plus coûteux qu'un appel à static_cast<>.
- On utilisera donc dynamic_cast<> seulement lorsqu'il n'y aura aucune autre solution.

- En conclusion, dynamic_cast<> est le seul opérateur de conversion à avoir un effet 'indéterminé' jusqu'à l'exécution.
- Son utilisation n'a du sens que lorsque confronté à du polymorphisme.
- Dans le cas où la conversion est assurée de réussir, on utilisera static_cast<>.

# const_cast<>

- Le const_cast<> permet de supprimer l'attribut constant d'une référence ou d'un pointeur (ex: const char* vers char*).
- C'est notamment le seul opérateur de conversions à pouvoir le faire.

Exemple const_cast<>

# reinterpret_cast< >

- L'opérateur reintepret_cast<> est le plus dangereux des opérateurs de conversions.
- Son rôle est de dire au compilateur: réinterprète-moi la représentation binaire de ce type en tant qu'un autre type.
- Il permet:
  1. De convertir n'importe quel type pointé en un autre, même lorsque ceux-ci n'ont aucun rapport (ex: int* vers double*).
  2. De convertir un type pointé en sa représentation intégrale et vice-versa (ex: int* vers int).

**Exemple reinterpret_cast<>**

# this

# Les membres static

- Normalement, les membres d'une classe n'ont d'existence que dans le contexte d'une instance de cette classe.
- Si une classe a comporte une variable de type int nommé **m_entier** et une fonction membre nommé **Fonction()**, il faut disposer d'une instance de la classe a pour être en mesure d'appeler **Fonction()** et d'utiliser la variable **m_entier**.
- Il existe cependant des membres qui dérogent de cette règle. Ce sont les membres **static**.

- Une variable membre statique est une variable qui existe toujours en un seul exemplaire, quel que soit le nombre d'instances de cette classe.
- Ces variables peuvent être exploitées de deux façons:
  1. Agir comme variable commune à toutes les instances, permettant de communiquer entre elles;
  2. Agir comme variable appartenant à la classe elle-même, et sans véritable lien avec les instances.
- Quoi qu'il en soit, l'existence des variables membres statiques n'est pas liée à l'instanciation de la classe.

- Les variables membres statiques sont déclarées, comme les autres, dans la définition de la classe.
- Leur seule différence est la présence du mot static devant la déclaration.

```
class A
{
public:
    int m_entier;   // declaration membre ordinaire
    static int m_entierStatic; // declaration membre static
}
```

- Lorsque la définition d'un type (ex: classe) ne crée aucune variable.
- Les variables membres 'ordinaires' ne sont créées que lorsque la classe est instanciée.
- Alors, la définition des variables membres 'ordinaires' se fait lors de l'instanciation de la classe.
- Cependant, quand est-ce que les variables membres static se font instancier?

- Puisque la création des variables membres statiques n'est pas reliée à l'instanciation (et ne doit pas être réitérée en cas d'instanciations multiples), il est donc nécessaire de créer les variables membres statiques en les définissant explicitement.
- Mais où les définir ces membres statiques?
- Dans le .h ou dans le .cpp?
- Quand?

- Il est impossible de redéfinir un objet qui existe déjà, nous le savons.
- Le fait que les fichiers .h sont amenés à faire l'objet de directives #include dans de multiples fichiers relevant de mêmes projets interdits donc d'y faire figurer des définitions.
- Donc, le lieu de la définition des variables membres statiques est le fichier .cpp qui contient la définition des fonctions membres, et non le fichier .h définissant la classe elle-même.

- La définition d'une variable membre statique exige la mention de son **nom complet,** mais le mot **static** ne doit pas être rappelé à cette occasion.
- Dans le cas de notre exemple, cette définition prendrait donc la forme suivante:

```
int A::m_entierStatic; // definition sans mention du mot static
```

- Il existe plusieurs façons d'accéder à une variable membre statique.
- L'une de ces méthodes est d'utiliser une instance de la classe pour accéder à la variable membre, comme s'il s'agissait d'une variable 'ordinaire':

```
A a; // instanciation de la classe
a.m_entierStatic = 15; // assignation variable static
```

- Cette méthode est cependant prescrite, puisque c'est la même variable, peu importe l'instance de la classe.
- Le recours à une instance pour accéder à une variable qui n'a aucun lien particulier avec elle donne une apparence trompeuse à un lecteur, par exemple.

- L'utilisation du this-> à l'intérieur de la fonction peut aussi être une méthode pour accéder à la variable statique.

```
class A
{
public:
    int m_entier;   // declaration membre ordinaire
    static int m_entierStatic; // declaration membre static

    A()
    {
        this->m_entierStatic = 10;
    }
};
```

- Même si cette façon de procéder est correcte, sa lisibilité est douteuse car le rappel du caractère statique de la variable n'est pas syntaxique mais dépend uniquement du choix judicieux du nom qui lui a été attribué.

- Dans tous les cas, il est préférable de toujours désigner les variables membres statiques par leur nom complet, ce qui élimine tout risque de malentendu quant à la nature de ces variables:

```
A::m_entierStatic = 40;
```

- La notation démontre clairement que la variable est statique.

Exemple Membre Static

Exemple Counter Instance Static

# Les fonctions static

- Les fonctions membres peuvent, elles aussi, être statiques!
- Les conséquences de ce statut sont toutefois légèrement différentes de celles que nous venons de voir, avec les variables membres statiques.
- Les fonctions membres statiques ne peuvent pas se différencier des fonctions 'ordinaires'.
- L'originalité d'une fonction membre statique réside dans le fait qu'il n'est pas nécessaire de disposer d'une instance pour appeler cette fonction.

- Comme dans le cas des variables membres, l'usage du nom complet d'une fonction membre permet de l'appeler, même en l'absence d'instances de la classe.
- Les appels sont identiques aux méthodes d'appels sur une variable statique.
- La déclaration d'une fonction membre statique se fait par la présence du mot static en avant de la déclaration:

```
static void fonctionStatic(); // declaration fonction statique
```

- Comme dans la définition d'une variable membre statique, le mot static ne doit pas figurer dans la définition d'une fonction membre statique:

```
void A::fonctionStatic()
{
    std::cout << "fonctionStatic" << std::endl;
}
```

- Les fonctions statiques ne disposent pas du paramètre caché this->, et ne peuvent donc pas accéder implicitement aux variables membres d'une instance.

- Les traitements effectués par une fonction membre statique ne peuvent donc pas accéder aux variables membres 'ordinaires' de la classe, et c'est logique.
- Donc, les fonctions membres statiques sont limitées aux variables membres statiques et aux objets auxquels leurs paramètres visibles leur donnent accès.

- En résumé:
  1. Les variables membres statiques sont des variables dont il n'existe pas un exemplaire dans chaque instance, mais un seul exemplaire qui peut être vu comme appartenant à la classe elle-même.
  2. Les variables membres statiques sont déclarées (comme les autres membres) dans la définition de la classe.
  3. À la différence des variables membres non statiques, les variables membres statiques doivent faire l'objet d'une définition explicite, ce qui offre une occasion de les initialiser facilement.
  4. Il est préférable de toujours désigner les variables membres statiques par leur nom complet, même lorsque le compilateur les reconnaîtrait à partir de leur nom abrégé.

5. Les fonctions membres statiques peuvent être invoquées sans référence à une instance particulière.
6. Comme les fonctions membres statiques ne sont pas invoquées au titre d'une instance, elles n'ont aucun accès privilégié aux variables membres d'une instance particulière. Les seules variables membres sur lesquelles elles peuvent opérer sont donc **les variables membres statiques**, et aux **variables membres d'objets locaux** ou rendus accessibles par des **paramètres**.

# LES POINTEURS INTELLIGENTS

- Pourquoi les pointeurs intelligents ont-ils été créés?
- Il existe trois problèmes reliés aux pointeurs classiques du C++:
  1. Ne pas libérer de la mémoire allouée dynamiquement.
  2. Libérer plusieurs fois de la mémoire allouée dynamiquement.
  3. Accéder à la avaleur pointée par un pointeur invalide.
- Les pointeurs intelligents sont là pour faciliter la vie aux programmeurs.
- Cependant, il faut tout de même porter une attention particulière sur l'utilisation de ceux-ci.

- Qu'est-ce qu'un pointeur intelligent?
- Pour supprimer les problèmes reliés aux pointeurs classiques, les pointeurs intelligents sont nés.
- Tout comme il existe différents scénarios d'utilisation des pointeurs, il existe aussi plusieurs types de pointeurs intelligents, selon le cas choisi.
- En fait, le pointeur intelligent est simplement une classe qui encapsule la notion d'une pointeurs.

- Les principaux problèmes reliés aux pointeurs classiques sont liés à la durée de vie des objets pointés, et au lien entre celle-ci et la durée de vie des pointeurs eux-mêmes
- C'est donc autour des opérations liées à la durée de vie des pointeurs (création, copie, destruction, etc.) qu'on ajoutera de l'intelligence.
- Le reste des opérations sont là pour donner un accès agréable au pointeur.

- En gros, un pointeur intelligent s'agit d'une classe que l'on utilise presque comme un pointeur, mais qui possède un mécanisme gérant la durée de vie des objets pointés.
- Cependant, il y a deux usages classiques des pointeurs qui ne sont pas couverts par les pointeurs intelligents:
  1. Les pointeurs sur des tableaux de caractères, ayant pour but de représenter des chaînes de caractères, pour lesquels les classe std::string et std::wstring apportent les mêmes avantages.
  2. Les pointeurs ayant pour but de gérer des tableaux de taille dynamique, rôle géré directement par les conteneurs de la bibliothèque standard, et en particulier std::vector.

- Il y a aussi certains cas pour lesquels des pointeurs intelligents sont moins indispensables.
- Par exemple, quand un objet appartient de manière unique et non équivoque à une classe qui s'occupe elle-même de la durée de vie de l'objet. (Ça vous rappelle quelque chose?)

- Le premier pointeur intelligent que nous allons voir sera probablement celui que vous utiliserez le plus: **shared_ptr<>**
- Le modèle derrière ce pointeur est celui où plusieurs pointeurs permettent l'accès à une même donnée, sans que l'un ce ceux-ci soit investi du rôle particulier consistant à être le responsable de la durée de vie de l'objet.
- Cette responsabilité est partagée entre tous les pointeurs pointant à un moment donné sur l'objet, d'où le mot '**shared**'.
- Comment dans ce cas la mémoire sera-t-elle libérée?

- En fait, quand un pointeur est détruit, il vérifie s'il n'est pas le dernier à pointeur sur l'objet.
- S'il est le dernier, l'objet est détruit et la mémoire est libérée.
- Comment cela fonctionne-t-il?
- Un compteur de référence est associé à l'objet. À chaque fois qu'un nouveau pointeur pointe sur l'objet, le compteur de référence est incrémenté, et à chaque fois qu'un pointeur arrête de pointer sur l'objet, (destruction du pointeur ou pointe sur un autre objet), le compteur est décrémenté.
- Si le compteur atteint 0, l'objet est détruit et la mémoire est libérée.

- Un **shared_ptr<>** est un template ayant comme paramètre le type d'élément sur lequel il doit pointer:

```
shared_ptr<int> a(new int (42));
```

- C'est une bonne habitude de stocker dès que possible un pointeur classique dans un pointeur intelligent.
- Cela s'assure qu'il n'aura pas l'occasion de créer une fuite de mémoire.
- Le standard a défini une fonction **make_shared<>** qui évite toute présence d'un pointeur classique:

```
shared_ptr<int> a = make_shared<int>(42);
```

- Cette fonction a aussi l'avantage d'être plus sûre et d'offrir un gain de performance.

Exemple shared_ptr<>

- Il existe un certain nombre de conversions possibles entre des pointeurs classiques.
- Les conversions équivalentes existent entre les **shared_ptr<>**.

```
class A {/*...*/};
class B : public A {/*...*/};

B* b1(new B);
A* a(b1);
B* b2 = dynamic_cast<B*>(a);

shared_ptr<B> sb1(new B);
shared_ptr<A> sa(sb1);
shared_ptr<B> sb2 = dynamic_pointer_cast<B>(sa);
```

- On peut noter en particulier que le mot clef **dynamic_cast** est remplacé par une fonction **dynamic_pointer_cast** qui s'utilise de manière semblable.
- Cette fonction retourne un pointeur ne pointant sur rien si la conversion n'a pu avoir lieu.
- Il existe de même une fonction **static_pointer_cast** et mimant l'effet d'un **static_cast** pour un pointeur classique.

- Il existe tout de même deux cas où malgré l'utilisation des pointeurs intelligents peuvent créer des fuites de mémoires.
- Le premier n'est pas lié aux shared_ptr tels quels, mais au fait que, temporairement, la mémoire n'a pas été gérée par des shared_ptr.

```
int f(shared_ptr<int> i, int j);
int g();

f(shared_ptr<int> (new int (42)), g());
```

```
int f(shared_ptr<int> i, int j);
int g();

f(shared_ptr<int> (new int (42)), g());
```

- Ce code respecte ce que nous avions précédemment dit, d'enrober au plus vite un pointeur classique dans un pointeur intelligent.
- Il présente cependant un risque de fuite de mémoire.
- Le compilateur voit:
  1. Créer un entier valant 42 dans une nouvelle zone mémoire allouée par un new
  2. Créer un shared_ptr avec le pointeur obtenu à l'étape 1
  3. Appeler la fonction g
  4. Appeler la fonction f
- Il y a une contrainte d'ordre entre ces opérations:
  1. L'étape 2 ne peut être appelée qu'après 1
  2. Et 4 seulement à la fin
- Mais le compilateur peut tout de même utilise l'ordre 1 3 2 4…

```
int f(shared_ptr<int> i, int j);
int g();

f(shared_ptr<int> (new int (42)), g());
```

- Donc, si la fonction g() plante, le **shared_ptr<>** en paramètre n'aura pas eu le temps de prendre possession de la mémoire, donc aucune possibilité de la libérer.
- Deux solutions à ce problème:
  1. Créer une variable temporaire:

```
int f(shared_ptr<int> i, int j);
int g();

shared_ptr<int> si (new int (42));
f(si, g());
```

  2. Soit ne pas allouer de pointeur classique du tout, et utiliser le **make_shared<>**:

```
int f(shared_ptr<int> i, int j);
int g();
f(make_shared<int>(42), g());
```

- L'autre risque de fuite de mémoire relié aux pointeurs intelligents est lors de la création d'un cycle.
- Examinons ce problème à l'aide d'un exemple:

Exemple cycle entre shared_ptr<>

- Il existe tout de même des inconvénients aux shared_ptr<>. Rien n'est parfait en programmation!
- Le premier inconvénient est que les shared_ptr<> ne gèrent pas les cycles.
- Un second inconvénient peut être les performances. Étant donné que les shared_ptr<> sont une couche par-dessus les pointeurs classiques, les performances en sont diminuées.
- Cependant, les pointeurs intelligents apportent des fonctionnalités très intéressantes par rapport aux pointeurs classiques:

- Il existe aussi des pointeurs intelligents nommés **weak_ptr<>**.
- Ils permettent de casser les cycles.
- Le principe est simple: parmi les pointeurs sur un objet, les **shared_ptr<>** se partagent la responsabilité de faire vivre ou mourir un objet, et les **weak_ptr<>** y ont un simple accès, sans aucune responsabilité associée.
- Un **weak_ptr<>** n'impacte donc pas le comptage de référence d'un objet.
- Quel est l'avantage alors d'un **weak_ptr<>** par rapport à un simple pointeur classique qui pointerait sur le même **shared_ptr<>**?

- La sécurité!
- Rappelez-vous: au moment où le compteur de référence d'un objet atteint 0, il est détruit.
- Si un autre pointeur essaye alors d'accéder à cet objet, c'est un comportement indéfini, car la variable sera null.

- Il est possible de créer un **weak_ptr<>** soit à partir d'un **shared_ptr<>**, soit à partir d'un autre **weak_ptr<>**.
- Il n'est cependant pas possible d'en créer un à partir d'un pointeur classique.
- Le **weak_ptr<>** n'est là que pour travailler en collaboration avec un **shared_ptr<>**.
- Il existe deux manières de travailler avec un weak_ptr<>:
  1. Créer un shared_ptr<> à partir du weak_ptr<> et de travailler à partir de ce pointeur.
  2. Utiliser la méthode lock() sur un weak_ptr.

- Il est possible de créer un **weak_ptr<>** soit à partir d'un **shared_ptr<>**, soit à partir d'un autre **weak_ptr<>**.
- Il n'est cependant pas possible d'en créer un à partir d'un pointeur classique.
- Le **weak_ptr<>** n'est là que pour travailler en collaboration avec un **shared_ptr<>**.
- Il existe deux manières de travailler avec un weak_ptr<>:
  1. Créer un shared_ptr<> à partir du weak_ptr<> et de travailler à partir de ce pointeur.
  2. Utiliser la méthode lock() sur un weak_ptr.

Exemple weak_ptr