# APP0
# Introduction à l'orienté objet C++

GUIDE DE L'ÉTUDIANT S2-
APP0 GEGI

*Hiver 2021 – Semaine 0*

*Document non-officiel*

Département de génie électrique et de génie informatique
Faculté de génie
Université de Sherbrooke
Pascal-Emmanuel Lachance

Lien pour classe Microsoft Teams
(https://teams.microsoft.com/l/channel/19%3ae79342f5927849c882a7d52c4bff89a0%40thread.tacv2/G%25C3%25A9n%25C3%25A9ral?groupId=0ebc411c-bd61-45af-9b43-53cbe532c084&tenantId=3a5a8744-5935-45f9-9423-b32c3a5de082)

Lien pour solution sur GitHub
(https://github.com/Raesangur/UdeS_S2_APP0)

Lien pour capsules vidéos pour le procédural
(https://www.youtube.com/playlist?list=PLGuJDRwFKGh3wlYMl6DscJhrmsN_d2BVl)

**Note** : En vue d'alléger le texte, le masculin est utilisé pour désigner les femmes et les hommes.

Document : Guide_Etudiant_2021_APP0_.docx
1re version décembre 2020
Rédigé par Pascal-Emmanuel Lachance
Révision et corrections par Samuel Martel

# TABLE DES MATIERES

# 1. ÉNONCÉ DE LA PROBLÉMATIQUE

La compagnie MesImages inc., très satisfaite de vos derniers services en traitement d'image, décide de vous réengager comme stagiaire, encore une fois dans le domaine de traitement des images. Leurs besoins ont changé depuis la dernière fois qu'ils eurent besoin de vos services, et ils ont maintenant besoin d'une nouvelle *codebase*, programmée en C++.

Ils n'ont pas besoin que vous réécriviez toutes les fonctions que vous leurs avez fournis durant votre dernier stage. Votre mandat est plutôt de créer une base sur laquelle ces fonctions pourraient être réimplémentées facilement. En effet, vous êtes chargés de créer une classe ImagePGM, et d'y implémenter quelques méthodes, dont les signatures vous sont fournies à l'Annexe A. Une de vos fonctions de manipulation d'image de votre dernier stage devra cependant y être implémentée, à des fins de tests.

La compagnie aurait également besoin de pouvoir ouvrir plusieurs tailles d'images différentes, et vous demande d'allouer dynamiquement l'espace mémoire nécessaire pour stocker l'image. Elle vous rappelle qu'il ne faut pas oublier de libérer l'espace mémoire alloué une fois que vous n'en aurez plus besoin, et vous suggère d'utiliser le destructeur de la classe pour cela. Vous aurez également à prendre un chemin de fichier sous la forme d'une *string* de la classe *std::string* de la librairie standard du C++, qui vous sera fournie par l'utilisateur au travers de l'invite de commande (vous devez donc gérer une entrée par l'utilisateur) (un pseudocode vous sera fourni pour cela). Finalement, ils veulent que vous puissiez comparer deux images à l'aide des opérateurs `==` et `!=`[1].

La compagnie vous suggère d'utiliser l'environnement Visual Studio Community 2019 pour le développement de votre projet, mais le choix d'IDE et d'outils vous revient.

Corrigé de la problématique disponible sur GitHub

[1] Par exemple :

```
1   ImagePGM image1{CHEMIN_IMAGE_1};
2   ImagePGM image2{CHEMIN_IMAGE_2};
3   if(image1 == image2)
4       ...
```

# 2. CONNAISSANCES NOUVELLES

**Connaissances déclaratives (QUOI?) :**

- Le langage C++
    - Classes
    - Références
    - *Namespaces* (espace de noms)
    - *Overloads* (surchage) de fonctions
    - *Templates* (modèles / patrons en bon français)
    - Constructeurs et destructeurs
    - Allocation de mémoire dynamique
    - Accesseurs
    - *Const*
- La librairie du C++
    - std::string
    - std::vector

**Connaissances procédurales (COMMENT?) :**

- Utiliser les classes pour traiter des groupes de données
- Allouer de la mémoire dynamiquement pour traiter des données diverses
- Construire des programmes orientés objets
- Utiliser la librairie du C++
- Utiliser des *templates* pour faciliter le développement de fonctions et méthodes
- Accéder aux membres d'une classe

**Connaissances conditionnelles (QUAND?) :**

- Développer une classe pour encapsuler le traitement d'un ensemble de données
- Allouer et gérer soi-même sa mémoire.

# 3. GUIDE DE LECTURE

## 3.1. Références essentielles

"C and C++ reference," *cppreference.com*. [Online]. Available: https://en.cppreference.com/w/. [Accessed: 20-Dec-2020].

"The C++ Resources Network," *cplusplus.com*. [Online]. Available: https://www.cplusplus.com/. [Accessed: 20-Dec-2020].

ISO/IEC. (2014). *ISO International Standard ISO/IEC 14882:2017(E) – Programming Language C++*. [Working draft]. Geneva, Switzerland: International Organization for Standardization (ISO). Retrieved from https://isocpp.org/std/the-standard. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf

## 3.2. Pour la création de classes

"C++ Classes and Objects," *GeeksforGeeks*, 08-Nov-2019. [Online]. Available: https://www.geeksforgeeks.org/c-classes-and-objects/. [Accessed: 20-Dec-2020].

*C++ Classes and Objects*. [Online]. Available: https://www.w3schools.com/cpp/cpp_classes.asp. [Accessed: 20-Dec-2020].

"C++ Classes and Objects," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm. [Accessed: 20-Dec-2020].

## 3.3. Références (versus pointeurs)

*Pointers vs References in C++*. [Online]. Available: https://www.tutorialspoint.com/pointers-vs-references-in-cplusplus. [Accessed: 21-Dec-2020].

"Pointers vs References in C++," *GeeksforGeeks*, 30-Jul-2019. [Online]. Available: https://www.geeksforgeeks.org/pointers-vs-references-cpp/. [Accessed: 21-Dec-2020].

*C++ Functions - Pass By Reference*. [Online]. Available: https://www.w3schools.com/cpp/cpp_function_reference.asp. [Accessed: 21-Dec-2020].

TheChernoProject, "REFERENCES in C++," *YouTube*, 18-Jun-2017. [Online]. Available: https://www.youtube.com/watch?v=IzoFn3dfsPA. [Accessed: 22-Dec-2020].

## 3.4. Namespaces

"Namespaces," *cplusplus.com*. [Online]. Available: https://www.cplusplus.com/doc/oldtutorial/namespaces/. [Accessed: 21-Dec-2020].

akbiggsakbiggs, "Why is 'using namespace std;" considered bad practice?," *Stack Overflow*, 01-Oct-1958. [Online]. Available: https://stackoverflow.com/questions/1452721/why-is-using-namespace-std-considered-bad-practice. [Accessed: 21-Dec-2020].

## 3.5. Surcharges et Templates

"Function Overloading in C++," *GeeksforGeeks*, 10-Dec-2018. [Online]. Available: https://www.geeksforgeeks.org/function-overloading-c/. [Accessed: 22-Dec-2020].

"operator overloading," *cppreference.com*. [Online]. Available: https://en.cppreference.com/w/cpp/language/operators. [Accessed: 28-Dec-2020].

"Templates," *cplusplus.com*. [Online]. Available: http://www.cplusplus.com/doc/oldtutorial/templates/. [Accessed: 21-Dec-2020].

"C++ Templates," *Programiz*. [Online]. Available: https://www.programiz.com/cpp-programming/templates. [Accessed: 21-Dec-2020].

TheChernoProject, "Templates in C++," *YouTube*, 08-Nov-2017. [Online]. Available: https://www.youtube.com/watch?v=I-hZkUa9mIs. [Accessed: 22-Dec-2020].

## 3.6. Constructeurs & Destructeurs

"Constructors in C++," *GeeksforGeeks*, 26-Oct-2020. [Online]. Available: https://www.geeksforgeeks.org/constructors-c/. [Accessed: 21-Dec-2020].

*C++ Constructors*. [Online]. Available: https://www.w3schools.com/cpp/cpp_constructors.asp. [Accessed: 21-Dec-2020].

"Destructors in C++," *GeeksforGeeks*, 26-Oct-2020. [Online]. Available: https://www.geeksforgeeks.org/destructors-c/. [Accessed: 21-Dec-2020].

"Constructors and member initializer lists," *cppreference.com*. [Online]. Available: https://en.cppreference.com/w/cpp/language/constructor. [Accessed: 25-Dec-2020].

TheChernoProject, "Copying and Copy Constructors in C++," *YouTube*, 13-Sep-2017. [Online]. Available: https://www.youtube.com/watch?v=BvR1Pgzzr38. [Accessed: 26-Dec-2020].

## 3.7. Allocation dynamique de mémoire

"new and delete operators in C++ for dynamic memory," *GeeksforGeeks*, 01-May-2020. [Online]. Available: https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/. [Accessed: 21-Dec-2020].

"C++ Dynamic Memory," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/cplusplus/cpp_dynamic_memory.htm. [Accessed: 21-Dec-2020].

gyanendra371, "Dynamic Constructor in C++ with Examples," *GeeksforGeeks*, 22-May-2019. [Online]. Available: https://www.geeksforgeeks.org/dynamic-constructor-in-c-with-examples/. [Accessed: 21-Dec-2020].

TheChernoProject, "The NEW Keyword in C++," *YouTube*, 01-Sep-2017. [Online]. Available: https://www.youtube.com/watch?v=NUZdUSqsCs4. [Accessed: 28-Dec-2020].

ReelLearning, "Pointers and Dynamic Memory in C++ (Memory Management)," *YouTube*, 18-Mar-2012. [Online]. Available: https://www.youtube.com/watch?v=CSVRA4_xOkw. [Accessed: 22-Dec-2020].

## 3.8. Const

Corob-Msft, "const (C++)," *Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/cpp/cpp/const-cpp?view=msvc-160. [Accessed: 26-Dec-2020].

"Const Correctness," *Const Correctness - C++ Tutorials - Cprogramming.com*. [Online]. Available: https://www.cprogramming.com/tutorial/const_correctness.html. [Accessed: 26-Dec-2020].

TheChernoProject, "CONST in C++," *YouTube*, 27-Aug-2017. [Online]. Available: https://www.youtube.com/watch?v=4fJBrditnJU. [Accessed: 26-Dec-2020].

# 4. LOGICIELS ET MATÉRIEL

- Visual Studio Community 2019 ou Visual Studio Community 2017, disponible sur https://visualstudio.microsoft.com/fr/downloads/

  Guide de setup de Visual Studio : https://youtu.be/CveHZHXIGC0

# 5. PRODUCTION À REMETTRE

Rien du tout.

# 6. ÉVALUATIONS

Rien non plus.

# 7. Pratique Procédurale

**Buts de l'activité**

- Comprendre les classes et leur utilisation
- Comprendre les constructeurs & destructeurs et leur utilisation
- Comprendre les *overloads* (surcharges) et leur utilisation
- Comprendre les templates et leur utilisation
- Comprendre l'allocation de mémoire dynamique
- Utiliser la classe `*std::string*` de la standard library

Voici le lien de la playlist contenant les capsules vidéo suivant tous les exercices.

https://youtube.com/playlist?list=PLGuJDRwFKGh3wlYMl6DscJhrmsN_d2BVl

## 7.1. Exercices – Classes

**Exercice 1.**

- Analyser le code suivant.
- Quelle est l'utilité d'utiliser une classe plutôt qu'une structure?
- Quelle est l'utilité du mot clé `*public*`? Qu'arriverait-il si on l'enlevait?
- Quelles sont les méthodes contenues dans ce programme?
- Donnez la signature complète de chaque fonction et méthode du programme.

```
1  #include <stdio.h>
2
3  class Personne
4  {
5  public:
6      char nom[128];
7      char prenom[128];
8      int  age;
9
10     void afficher()
11     {
12         printf("Nom: %s\n",nom);
13         printf("Prenom: %s\n",prenom);
14         printf("Age: %d\n",age);
15     }
16 };
17
18
19 int main()
20 {
21     Personne p;
22
23     printf("Nom: ");
24     scanf("%s",p.nom);
25     printf("Prenom: ");
26     scanf("%s",p.prenom);
27     printf("Age: ");
28     scanf("%d",&p.age);
29
30     p.afficher();
31
32     return 0;
33 }
```

## 7.2. Exercices – Constructeurs & Destructeurs

**Exercice 2.**

- Analyser le code suivant.
- Qui sera supprimé en premier?
- Quelle sera la sortie de ce programme?
- Pourquoi est-ce que la méthode `*c_str*` de `*std::string*` est appellée?
- Quelle est la différence entre `*stdio.h*` et `*cstdio*`?
- Qu'est-ce qui aurait pu être utilisé au lieu de `*printf*`?
- Est-ce qu'on aurait pu définir un destructeur qui prenait un paramètre?

```
1  #include <cstdio>
2  #include <string>
3
4  class Personne
5  {
6  public:
7      std::string nom;
8
9      Personne(std::string nomPersonne)
10     {
11         nom = nomPersonne;
12
13         printf("Bon matin %s\n", nom.c_str());
14     }
15     ~Personne()
16     {
17         printf("Bonne fin de matin %s\n", nom.c_str());
18     }
19 };
20
21 int main()
22 {
23     Personne george("George");
24     Personne jonathan("Jonathan");
25
26     if (true)
27     {
28         Personne emilia("Emilia");
29     }
30
31     return 0;
32 }
```

## Exercice 3.

- Analyser le code suivant.
- Quelles sont les deux différences avec le code précédent?

```
1  #include <cstdio>
2  #include <string>
3
4  class Personne
5  {
6  public:
7      std::string nom;
8
9      Personne(std::string nomPersonne) : nom(nomPersonne)
10     {
11         printf("Bon matin %s\n", nom.c_str());
12     }
13     ~Personne()
14     {
15         printf("Bonne fin de matin %s\n", nom.c_str());
16     }
17 };
18
19
20 int main()
21 {
22     Personne george{"George"};
23     Personne jonathan("Jonathan");
24
25     if(true)
26     {
27         Personne emilia{"Emilia"};
28     }
29
30     return 0;
31 }
32
```

## 7.3. Exercices – Références

**Exercice 4.**

- Analyser le code suivant
- Aurait-il été possible de nommer la classe `*string*` au lieu de `*myString*`?
- Si on avait rajouté, à la ligne 5, la ligne suivante : `*using namespace std;*`
  - Aurait-il été possible de nommer la classe `*string*` au lieu de `*myString*`?
  - Aurait-on pu enlever le `*std::*` de la ligne 9 (`*std::string str;*`)
  - Donnez des exemples de noms de variables / fonctions pouvant alors causer des conflits.
- À quoi servent les deux constructeurs de la classe `*myString*`?
- Combien de fois le destructeur de la classe est-il appelé durant l'exécution de ce programme?
- Est-ce que l'assignation à la ligne 27. modifie la string dans `merveilleuxMessage`?
- Quelle est la sortie de ce programme?

```
1  #include <cstdio>
2  #include <string>
3
4  //using namespace std;
5
6  class myString
7  {
8  public:
9      std::string str;
10     myString(const char* text) : str{text}
11     {
12         printf("Creation de la string\n");
13     }
14     myString(myString& copiedString) : str{copiedString.str}
15     {
16         printf("Copie de la string\n");
17     }
18     ~myString()
19     {
20         printf("Destruction de la string\n");
21     }
22 };
23
24 void utiliserString(myString str)
25 {
26     printf("Utilisation de la string\n");
27     str.str = "";
28 }
29
30 int main()
31 {
32     myString merveilleuxMessage{"Bon matin"};
33
34     utiliserString(merveilleuxMessage);
35     printf("%s\n", merveilleuxMessage.str.c_str());
36
37     return 0;
38 }
```

## Exercice 5.

- Analyser le code suivant
- Quelle est la différence avec le code précédent?
- Quelle sera la sortie du programme?
- Aurait-il été possible de remplacer la référence par un pointeur?
- Quelle est l'utilité des références?

```
1  #include <cstdio>
2  #include <iostream>
3  #include <string>
4
5  class myString
6  {
7  public:
8      std::string str;
9      myString(const char* text) : str{text}
10     {
11         printf("Creation de la string\n");
12     }
13     myString(myString& copiedString) : str{copiedString.str}
14     {
15         printf("Copie de la string\n");
16     }
17     ~myString()
18     {
19         printf("Destruction de la string\n");
20     }
21 };
22
23 void utiliserString(myString& str)
24 {
25     printf("Utilisation de la string\n");
26     str.str = "";
27 }
28
29 int main()
30 {
31     myString merveilleuxMessage{"Bon matin"};
32
33     utiliserString(merveilleuxMessage);
34     printf("%s\n", merveilleuxMessage.str.c_str());
35
36     return 0;
37 }
```

## 7.4. Exercices – Allocation de mémoire dynamique

### Exercice 8.

- Analyser le code suivant.
- Qu'est-ce que la fonction `*fonction*` retourne?
- Y-a-t'il une erreur de conception dans ce programme?
- Quelle est la sortie du programme?

```
1  #include <cstdio>
2
3  int* fonction(int parametre)
4  {
5      int chiffreCool = parametre;
6      return &chiffreCool;
7  }
8
9  int main()
10 {
11     int* pa = fonction(42);
12     int a = *pa;
13
14     printf("%d\n", a);
15     return 0;
16 }
```

### Exercice 9.

- Analyser le code suivant.
- Y-a-t'il une erreur de conception dans le programme?
- Quelle est la sortie du programme?
- Quelle est l'utilité des opérateurs `*new*` et `*delete*`?
- Que se passe-t'il si nous omettons `*delete*` à la ligne 18?
- Quel est le type de variable alloué par `new int`?
-

```
1  #include <cstdio>
2
3  int* fonction(int parametre)
4  {
5      int* chiffreCool = new int;
6      *chiffreCool = parametre;
7
8      return chiffreCool;
9  }
10
11 int main()
12 {
13     int* pa = fonction(42);
14     int a = *pa;
15
16     printf("%d\n", a);
17
18     delete pa;
19     return 0;
20 }
```

## Exercice 10.

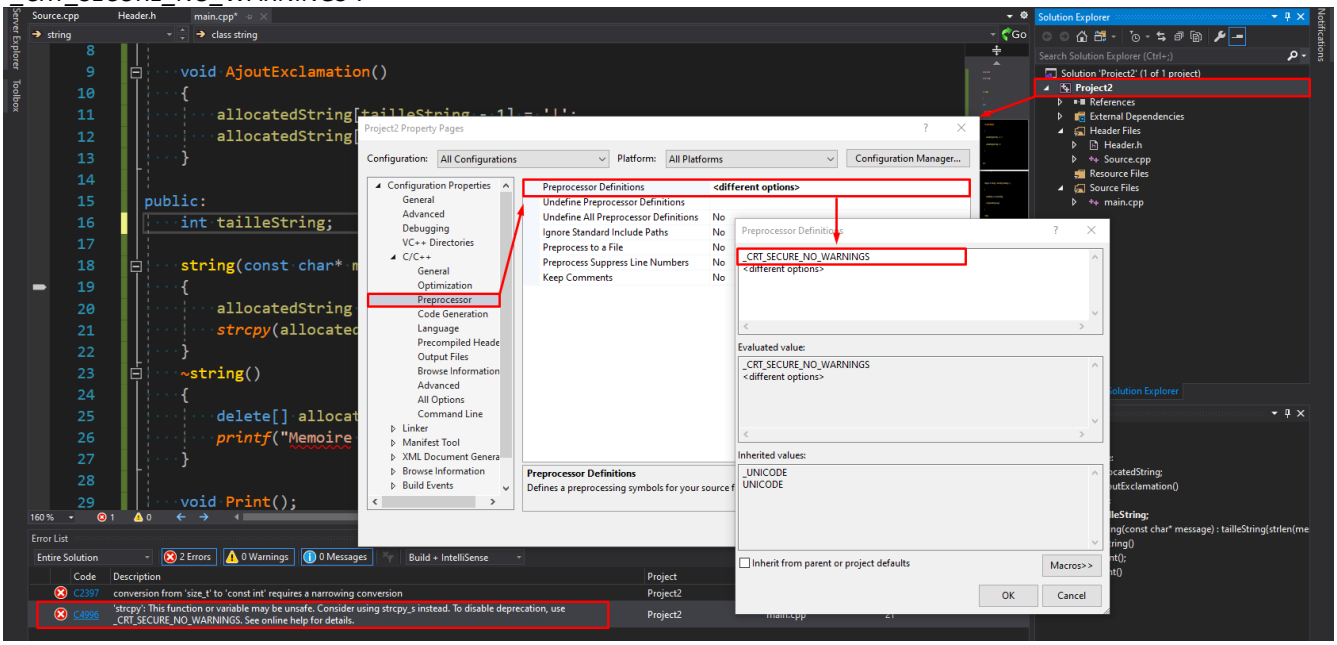
- Analyser le code suivant
- Quelle est le type de variable alloué par `*new char[tailleMessage + 1]*`
- À quoi servent les `*+1*` insérés dans `*tailleMessage*` et dans l'allocation?
- Quelle est la sortie du programme?
- Quelle est la différence entre `*delete*` et `*delete[]*`?
- Le programme compile-t'il si `*delete*` est utilisé à la place de `*delete[]*`?

```
1  #include <cstdio>
2  #include <cstring>
3
4  int main()
5  {
6      const char* merveilleuxMessage = "Bon matin";
7
8      int tailleMessage = strlen(merveilleuxMessage) + 1;
9      char* coolString = new char[tailleMessage + 1];
10
11     for (int i = 0; i < tailleMessage; i++)
12     {
13         coolString[i] = merveilleuxMessage[i];
14     }
15     coolString[tailleMessage - 1] = '!';
16     coolString[tailleMessage] = '\0';
17
18     printf("%s\n", coolString);
19     delete[] coolString;
20     return 0;
21 }
```

**Exercice 11**.

- Sans regarder le code à la page suivante, écrire une classe qui sert à rajouter un point d'exclamation à un message.
  - Son constructeur doit prendre un `*const char**`, une string C-style avec un `\0` de fin.
  - Son constructeur doit allouer l'espace mémoire nécessaire à l'opération.
  - Son constructeur doit copier la string passée en paramètre dans l'espace mémoire alloué. (la fonction `*strcpy*` de la librairie du C peut être utilisée).[2]
  - Elle doit contenir une méthode privée `*AjoutExclamation*`, qui est appelée dans le constructeur
  - Elle doit contenir un champ `*tailleString*`, constant, et public. Vous pouvez également rendre ce champ privé et le remplacer par un accesseur.
  - La mémoire allouée par le constructeur doit se trouver dans un champ privé de la classe.
  - Le champ `*tailleString*` doit être initialisé par une `*member initialisation list*`.[3]
  - Elle doit contenir une méthode publique `*Print*`, qui imprime la string modifiée.
  - La déclaration de la méthode `*Print*` doit être dans le corps de la classe, mais sa définition doit être en dehors du corps de la classe.
  - Elle doit libérer automatiquement sa mémoire une fois qu'on en a plus besoin.
  - Un message de confirmation de la libération de la mémoire doit s'afficher.

[2] Si Visual Studio n'aime pas l'utilisation de la fonction `*strcpy*`, vous pouvez rajouter la définition `*_CRT_SECURE_NO_WARNINGS*`.




[3] Si Visual Studio n'aime pas l'assignation dans `*tailleString*` depuis la fonction `*strlen*`, c'est parce que `*strlen*` retourne un `*size_t*` et non pas un `*int*`. Il est soit possible de donner à `*tailleString*` le type `*size_t*`, ou possible de faire un `*cast*` vers `*int*`.

## Solution Exercice 11.

```cpp
#include <cstdio>
#include <cstring>

class string
{
private:
    char* allocatedString;

    void AjoutExclamation()
    {
        allocatedString[tailleString - 2] = '!';
        allocatedString[tailleString - 1] = '\0';
    }

public:
    const int tailleString;

    string(const char* message) : tailleString{(int)strlen(message) + 2}
    {
        allocatedString = new char[tailleString];
        strcpy(allocatedString, message);

        AjoutExclamation();
    }
    ~string()
    {
        delete[] allocatedString;
        printf("Memoire libre maintenant\n");
    }

    void Print();
};

void string::Print()
{
    printf("%s\n", allocatedString);
}

int main()
{
    string merveilleuxMessage{"Bon matin"};

    merveilleuxMessage.Print();

    return 0;
}
```

## 7.5. Exercices – Accesseurs et mutateurs (getters & setters)

**Exercice 12.**

- Analyser le code suivant
- Est-ce que la valeur retournée par la méthode `*Get_Nom*` est la même que `*_nom*`? Est-ce une copie de la valeur ou une référence?
- Est-il possible de modifier `*_nom*` à partir de `*nomDeBob*`?
- Est-ce que `*__nom*` ou `*_Nom*` seraient des noms acceptables par convention?
- Combien de `*std::string*` sont créées au court de l'exécution de ce programme (sans optimisation).
- Serait-il possible de rendre ce code plus rapide et optimal?
- Comment s'assurer que ce qui est pointé par une référence ne se fasse pas modifier?

```
1  #include <cstdio>
2  #include <string>
3
4  class Personne
5  {
6  private:
7      std::string _nom;
8
9  public:
10     Personne(std::string nom) : _nom{nom}
11     {
12     }
13
14     std::string Get_Nom()
15     {
16         return nom;
17     }
18 };
19
20 int main()
21 {
22     Personne bob("Bob");
23
24     std::string nomDeBob = bob.Get_Nom();
25     printf("%s\n", nomDeBob.c_str());
26
27     return 0;
28 }
```

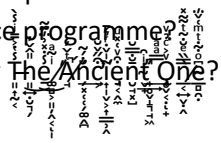
## Exercice 13

- Analyser le code suivant
- Trouver les différences par rapport au code de l'exercice 12
- Combien de `*std::string*` sont créées au court de l'exécution de ce programme?
- Aurait-on pu omettre le `*const*` à la ligne 24 si on ne prenait pas une référence?
- Quel est l'avantage de passer par référence par rapport à passer par copie?
- Est-il possible de modifier `*_nom*` à partir de `*nomDeBob*`?
- L'utilisation du mot-clé `*const*` implique-il des restrictions d'accès au-delà de l'étape de compilation? (Autrement dit, est-ce qu'indiquer qu'une variable est *const* est uniquement se mettre des bâtons dans les jambes pour le compilateur, où y-a-t'il des différences au niveau de l'assembleur généré)

```
1  #include <cstdio>
2  #include <string>
3
4  class Personne
5  {
6  private:
7      std::string _nom;
8
9  public:
10     Personne(const std::string& nom) : _nom{nom}
11     {
12     }
13
14     const std::string& Get_Nom()
15     {
16         return _nom;
17     }
18 };
19
20 int main()
21 {
22     Personne bob("Bob");
23
24     const std::string& nomDeBob = bob.Get_Nom();
25     printf("%s\n", nomDeBob.c_str());
26
27     return 0;
28 }
```

## Exercice 14.

- Analyser le code suivant
- Quelle est l'utilisé du mutateur (*setter*) dans ce cas-ci?
- Donnez quelques utilités possibles à l'utilisation d'un accesseur (*getter*).
- Donnez quelques utilités possibles à l'utilisation d'un mutateur (*setter*).
- Quelle est la sortie de ce programme?
- A-t-on évité d'invoquer The Ancient One?



```
1  #include <cstdio>
2  #include <string>
3
4  class Personne
5  {
6  private:
7      std::string _nom;
8
9  public:
10     Personne(const std::string& nom) : _nom{nom}
11     {
12     }
13
14     const std::string& Get_Nom()
15     {
16         return _nom;
17     }
18     void Set_Nom(const std::string& nouveauNom)
19     {
20         if (nouveauNom != "cthulhu")
21         {
22             _nom = nouveauNom;
23         }
24     }
25 };
26
27 int main()
28 {
29     Personne bob("Bob");
30
31     bob.Set_Nom("Bobinette");
32
33     bob.Set_Nom("cthulhu");
34
35     const std::string& nomDeBob = bob.Get_Nom();
36     printf("%s\n", nomDeBob.c_str());
37
38     return 0;
39 }
```

## 7.6. Exercices – *std::string*

Documentation de la classe *std::string sur cppreference.com*

**Exercice 15.**

- Analyser le code suivant
- Y-a-t'il une différence entre l'initialisation de la string à la ligne 8 et celle à la ligne 9?
- Que fait la méthode `*std::string::empty()*`?
- Donnez deux façons de supprimer tout le contenu d'une `*std::string*`.
- Qu'est-ce qu'une concaténation?
- Que fait l'opérateur *+* entre deux `*std::string*`?
- Quelle est la différence entre la méthode `*std::string::append*` et l'opérateur *+=* entre deux `*std::string*`.
- Quel est le type de variable retourné par `*std::to_string*` à la ligne 20?
- Que fait le mot-clé `*auto*`?
- Aurait-on pu faire `*phrase.append(annee)*` directement?
- Est-il possible d'utiliser `*std ::to_string*` avec des valeurs à virgules flottantes? Avec des `*std::string*`?
- Quelle est la différence entre la méthode `*std::string::c_str*` et la méthode `*std::string::data*`.
- Pour interfacer avec la librairie `*<string.h>*` (ou `*<cstring>*`), vaut-il mieux utiliser `*std::string::c_str*` ou `*std::string::data*`?
- Quelle est la sortie de ce programme?

```
1  #include <cstdio>
2  #include <string>
3
4  int main()
5  {
6      const char* phraseDeBase = " inventa le langage C en ";
7
8      std::string prenom{"Dennis"};
9      std::string nomFamille = "Ritchie";
10     int annee = 1972;
11
12     std::string phrase;
13     if(phrase.empty() == true)
14     {
15         printf("La phrase est vide\n");
16     }
17     phrase = prenom + ' ' + nomFamille;
18     phrase.append(phraseDeBase);
19
20     auto anneeString = std::to_string(annee);
21     phrase.append(anneeString);
22
23     printf("%s",phrase.c_str());
24     printf("Pointeur1: %p, Pointeur2: %p", phrase.c_str(), phrase.data());
25     return 0;
26 }
```

## 7.7. Exercices – Overloads (surcharges)

**Exercice 6.**

- Analyser le code suivant.
- Est-ce que ce code compile en C? Compile-t'il en C++?
- Laquelle des deux versions de `*add*` est-ce que l'assignation à la ligne 16 appelle?
- Quelle est la sortie de ce programme?
- Comment le compilateur fait-il la différence entre les deux fonctions?
- Comment faire l'équivalent en C (sans macros)? Faudrait-il deux fonctions avec des noms différents?

```
1  #include <cstdio>
2
3  int add(int parametre1, int paramtre2)
4  {
5      return parametre1 + paramtre2;
6  }
7  double add(double parametre1, double parametre2)
8  {
9      return parametre1 + parametre2;
10 }
11
12 int main()
13 {
14     int a = add(1, 1);
15     double b = add(1.0, 1.0);
16     int c = add(3.99, 0.99);
17
18     printf("a: %d, b: %f, c: %d", a, b, c);
19     return 0;
20 }
```

## Exercice 16.

## (sur cppInsights)

- Analyser le code suivant
- Qu'est-ce qui a été surchargé dans la classe `*Personne*`?
- Qu'est-ce qui est comparé?
- Qu'est-ce qui est additionné?
- Dans quel ordre est-ce que les destructeurs sont appelés?
- Quelle est la sortie de ce programme?
- Donnez un exemple d'un opérateur surchargé dans la librairie du C++
- Serait-il possible de créer une surcharge qui prend une `*std::string*` en paramètre?
- Est-ce que l'opérateur `*+=*` et l'opérateur `*+*` doivent prendre des surcharges différentes?
- Que se passe-t'il si on ne précise pas de surcharge pour l'opérateur de comparaison et qu'on cherche à comparer deux objets d'une classe?
  a) Tous les membres de la classe sont comparés ensemble
  b) Le code ne compile pas
  c) Le comportement n'est pas défini (n'importe quel résultat)
  d) Il compare uniquement le premier membre de chaque classe
  e) Il suffit qu'un seul des membres de la classe soit égal
  f) Le code compile, mais lance une exception en exécution

- Créez l'opérateur `*!=*` de la classe `*Personne*` à partir de l'opérateur `*==*`.
- Trouvez les deux opérateurs qui ne peuvent pas être surchargés :

| | | | | | |
|---|---|---|---|---|---|
| **Comparaison d'égalité** | == | **Assignation** | = | **OU logique** | \|\| |
| **Opérateur ternaire** | ?: | **Allocation de mémoire** | *new* | **Accès de membre** | -> |
| **Déréférenciation** | * | **Indexation** | [ ] | **Déplacement** | >> |
| **Comparaison trilatérale** | <=> | **Suffixe** | " " | **Taille d'un objet** | *sizeof* |

```cpp
#include <cstdio>
#include <string>

class Personne
{
private:
    std::string m_nom;
    int m_age;
    float m_salaireHoraire;

public:
    Personne(const std::string& nom, int age, float salaire) : m_nom{nom}, m_age{age}, m_salaireHoraire{salaire}
    {
        printf("Bon matin %s\n", m_nom.c_str());
    }

    bool operator==(const Personne& autrePersonne)
    {
        return (autrePersonne.m_nom == m_nom) && (autrePersonne.m_age == m_age);
    }
    Personne operator+(const Personne& autrePersonne)
    {
        return Personne{m_nom, m_age + autrePersonne.m_age, m_salaireHoraire + autrePersonne.m_salaireHoraire};
    }
};

int main()
{
    Personne jonathan1{"Jonathan", 21, 15.75};
    Personne jonathan2{"Jonathan", 21, 20.50};
    Personne phillipe{"Phillipe", 47, 45.00};

    if (jonathan1 == jonathan2)
    {
        printf("Les deux Jonathans sont egaux!\n");
    }

    if (jonathan1 == phillipe)
    {
        printf("Jonathan et Phillipe sont la meme personne?\n");
    }

    Personne megaJonathan = jonathan1 + phillipe;

    return 0;
}

```

## 7.8. Exercices – Templates

**Exercice 7.**

**(sur cppInsights)**

- Analyser le code suivant
- À quoi sert une template?
- Quelle est un autre mot clé équivalent à `*typename*` dans ce contexte?
- Est-ce que le code généré par le compilateur sera différent du codé généré pour l'exercice 6 (section 7.7).
- Peut-on templater des méthodes dans une classe?
- Peut-on templater une classe elle-même?

```
1  #include <cstdio>
2
3  template <typename monType>
4  monType add(monType parametre1, monType parametre2)
5  {
6      return parametre1 + parametre2;
7  }
8
9  int main()
10 {
11     int a = add(1, 1);
12     double b = add(1.0, 1.0);
13     int c = add(3.99, 0.99);
14
15     printf("a: %d, b: %f, c: %d", a, b, c);
16     return 0;
17 }
```

# 8. Annexe A : Signature des fonctions et méthodes à implémenter pour la problématique

## 8.1. Fichiers

Le code que vous remettez devrait contenir les trois fichiers suivants :

- main.cpp
- classeImagePGM.hpp
- classeImagePGM.cpp

### main.cpp

Ce fichier devra contenir une simple fonction main, qui demande à l'utilisateur :

- d'entrer le chemin d'un fichier. Vous n'aurez pas à faire la gestion d'erreurs (si le fichier n'existe pas, ect).
- De confirmer l'opération à exécuter sur l'image chargée
- De rentrer le chemin d'écriture de la nouvelle image maintenant modifiée

Il vous est recommandé de créer plusieurs fonctions dans le fichier main.cpp pour encapsuler et modulariser les tâches.

### classeImagePGM.hpp

Ce fichier devra contenir la définition de la classe ImagePGM.
Cette définition contiendra les différents champs membres de la classe, ainsi que la déclaration de ses méthodes.
Aucune définition de méthode ne devra se trouver dans ce fichier.

### classeImagePGM.cpp

Ce fichier contiendra toutes les définitions des méthodes déclarées dans le fichier classeImagePGM.hpp

## 8.2. Champs

Tous les champs privés de la classe commenceront, par convention, par le préfixe `*m_*`.

Le type `*uint16_t*` est défini dans `*<stdint.h>*` (ou `*<cstdint>*`), et représente un entier non-signé à 16-bits.

L'utilisation de `*unsigned short*` est également permise.

```
1  private:
2      const std::string m_nomFichier;
3      uint16_t*         m_image   = nullptr;
4      uint16_t          m_largeur = 0;
5      uint16_t          m_hauteur = 0;
6      uint16_t          m_maxVal  = 0;
7
8      // Optionnel, seulement si vous voulez garder les commentaires d'en-tête
9      std::vector<std::string> m_commentaires;
```

## 8.3. Méthodes

Les méthodes indiquées dans cette section sont les méthodes qu'il faut strictement faire, cependant, il vous est fortement recommandé de créer d'autres petites méthodes (généralement privées), pour séparer des grosses méthodes en plus petites.

### Constructeurs

```
1   public:
2       ImagePGM() = default;
3       ImagePGM(const std::string& nomFichier);
4       ImagePGM(const ImagePGM& autreImage);
5       ~ImagePGM();
```

#### *Constructeur défaut*

Le constructeur par défaut permet de créer une instance de classe sans y spécifier d'arguments. On peut soit le créer vide :

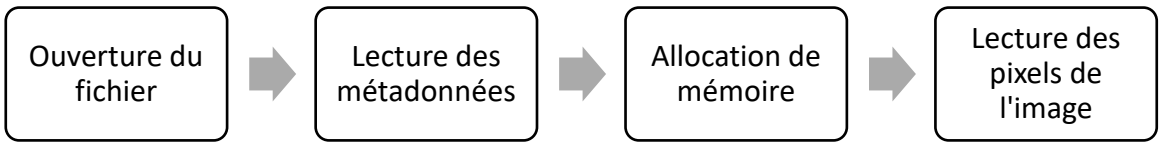
```
1   ImagePGM()
2   {
3   }
```

Ou utiliser le mot clé `*default*` qui laisse le compilateur s'en charger.
Le constructeur par défaut n'est pas requis, mais fortement recommandé.

#### *Le constructeur par fichier*

Ceci est le constructeur le plus important. Le constructeur par fichier va lire un fichier spécifié par la `*std::string*` reçue en argument, et charger ses pixels et métadonnées.
Elle doit allouer l'espace nécessaire en lisant la taille de l'image dans les métadonnées.
Un *buffer* (tampon) de lecture de 128 octets peut être utilisé pour la lecture, bien qu'il soit recommandé de faire l'allocation de ce buffer-là également durant le constructeur (bien entendu, il faut également libérer la mémoire du buffer), ou de bypasser entièrement l'utilisation d'un buffer pour la lecture.

Un algorithme très simple vous a été fourni pour ce constructeur.




#### *Constructeur par copie*

Ce constructeur permet de faire une copie conforme des données d'une autre image.
Il faut faire attention à ne pas simplement copier le pointeur d'une autre image; il faut allouer un nouvel espace mémoire et y copier tous les nouveaux pixels.

#### *Destructeur*

Une fois qu'on a plus besoin d'une image, il faut libérer sa mémoire allouée d'une façon ou d'une autre.

### Méthodes privées

```
1   private:
2       uint16_t& Pixel(uint16_t ligne, uint16_t colonne) const;
```

Une seule méthode privée est demandée (bien qu'en faire plus pour segmenter votre code est recommandé).

Il s'agit simplement d'une fonction d'accès en lecture d'un pixel. Les opérations matricielles bi-dimensionnelles n'étant pas possibles sur des simples pointeurs, la fonction Pixel est un `*wrapper*` autour d'une simulation de cette opération.

L'image n'as pas de dimensions de base sous forme de tableau bi-dimensionnel. Nous devons donc simuler ces dimensions.
Un tableau `*int x[3][5]*` est un tableau de 3 `*int[5]*` successifs. Se déplacer dans le tableau à l'aide d'un pointeur nécessite de déplacer le pointeur par la taille d'un `*int[5]*`, et non pas un simple `*int*`.
L'accès à un pixel spécifique pourrait donc être implémenté de la manière suivante :

```
pixel = image[ligne * largeur + colonne]
```

### Interfaçage

```
1   public:
2       void Sauvegarde(const std::string& nomFichier);
3       void Imprime();
```

Ces deux fonctions d'interfaçages permettent de sauvegarder à un fichier spécifié (créer le fichier si nécessaire) sous format PGM, et d'imprimer à l'écran toutes les valeurs des pixels de l'image.
Si désiré, la fonction de sauvegarde réécrit les commentaires d'en-tête originellement lus.

### Comparaison

```
1   public:
2       bool operator==(const ImagePGM& autreImage);
3       bool operator!=(const ImagePGM& autreImage);
```

Ces deux surcharges d'opérateurs permettent de vérifier si deux instances d'ImagePGM sont pareilles. Elle doivent comparer pixel par pixel les deux images.

### Accesseurs

```
1   public:
2       const std::string& Get_NomFichier();
3       uint16_t           Get_Hauteur();
4       uint16_t           Get_Largeur();
5       uint16_t           Get_MaxVal();
```

Ces quatre fonctions d'accès permettent de récupérer les métadonnées de l'image.

## Opérations sur les images

```
1       // Fonctions à réimplémenter en C++.
2       // Une seule de ces fonctions devra être faite dans le cadre de cet APP0.
3   public:
4       void                    CreerNegatif();
5       std::vector<uint16_t>   CreerHistogramme();
6       uint16_t                CouleurPreponderante();
7       void                    EclaircirNoircir(int32_t valeur);
8       void                    Pivoter90(bool sensHoraire);
9       ImagePGM                Extraire(int ligneCoin1, int colonneCoin1,
10                                       int ligneCoin2, int colonneCoin2);
```

Ces opérations sont les mêmes que vous avez eues à réaliser au cours de votre premier stage chez MesImages inc. Vous n'en avez qu'une seule à reprogrammer au cours de cette problématique, pour démontrer le fonctionnement de votre classe.

Il y a eu certains ajustements à faire à certaines de ces opérations. (Si rien n'est spécifié ci-dessous, l'opération reste inchangée).

### *Créer histogramme*

L'histogramme des intensités lumineuses doit maintenant être stockées dans un `*std::vector*` (un tableau à taille variable). On doit donc y réserver la bonne taille, et puis retourner le vecteur. (Il n'y a pas d'allocation ou de désallocations manuelles à faire, c'est fait pour vous dans la librairie du C++).

### *Extraire une sous-image*

Vous devez être capable de créer une sous-image à partir de votre image originelle, sans affecter l'objet originel. Cette sous-image devra par la suite être retournée.

C'est pour cette fonction qu'il vous est recommandé d'implémenter un constructeur par défaut, qui ne fait rien, puis, il vous suffira d'y copier les métadonnées et les pixels désirés (sans oublier de faire une allocation de mémoire).