# PoP C++ Série 4

## H1 :
## a) usage de static à l'échelle d'un module
## b) exercises on graphs

## H2 : MOOC [MOOC Introduction à la programmation orientée objet (en C++)]
### Série semaine 3: constructeur/ destructeur
### Surcharge des opérateurs

---

# Première partie :

# a) static à l'échelle d'un module

**Exercice 1.1**(niveau 0) : **module student gérant un ensemble de variables Student.**

Le document séparé illustre l'usage de **static** pour un type concret mis en œuvre avec une structure.

**Exercice 1.2 (niveau 2) : gestion d'un agenda**

On testera les modules **date** et **event** au fur et à mesure des questions avec un module de niveau supérieur **test** qui inclut leur interface et appelle les fonctions exportées.

a) Méthode de travail : Commencer par écrire le fichier **makefile** et les **fichiers .cc et .h dans une version minimale** qui compile pour produire l'exécutable **test**, avant de préciser les classes **date** et **event** avec les questions suivantes. On mettra en oeuvre les header guard pour les interfaces des modules (série3 PoP).

b) module **date** gérant une classe **Date** de calendrier
Ecrire un module responsable d'une classe Date avec 3 attributs privés : **jour, mois, annee.**

Un seul constructeur sera défini pour initialiser simultanément les 3 champs en vérifiant que la date est correcte pour le calendrier grégorien (la série1 fourni le code source qui effectue ces vérifications). Un message est affiché en cas d'erreur.

c) Une méthode publique de calcul du *nombre de jours entre deux dates* sera aussi fournie (cf série1).

d) Ecrire une surcharge des opérateurs == et != pour la classe **Date**

e) module **event** gérant une classe **Event** et un ensemble d'instance de ce type dans un tableau dynamique **agenda** restant confidentiel au niveau du module event.
La classe Event contient les attributs privés : **nom** et **lieu** de type string, et **date** de type Date.

Un constructeur doit permettre d’initialiser une instance à partir de la valeur de tous les attributs.

f) Ecrire une surcharge des opérateurs == et != pour la classe **Event**

g) Une méthode publique **add_to_agenda** doit permettre d’ajouter la valeur d’une variable de type Event à l’ensemble des Event mémorisés dans **agenda** au niveau du module **event**. Les conditions à remplir sont les suivantes :
- L’agenda ne doit pas contenir déjà cet **Event**
- il ne doit pas y avoir d’Event déjà prévu pour le même lieu à la même *date plus ou moins 6 jours*
Un message d’erreur est affiché en cas d’erreur.

h) Une méthode publique doit permettre d’afficher l’Event sur lequel elle est appelée.
Une méthode de classe doit permettre d’afficher l’ensemble des Event mémorisés dans l’agenda.

---

# b) exercises on graphs (in english)

The exercise this week shows you how to implement a graph and some simple algorithms in C++. Please be aware that the code of this exercise is not entirely compliant with your styleguide. Before you use it in your project, you'll have to refactor it.

## ex b.1) (niveau 1)

We start with a small class for a **Node** visible on next page. It contains an **id** string and a list of **neighbours**. Each neighbour is represented by an integer, which is an index in a static **vector<Node>** named **nodes**.

We make an important assumption here: all nodes in our graph are stored in **vector<Node> nodes** and they are *never deleted*. This means that we can use the index of a node in this vector as a unique identifier. Each node keeps track of a list of its neighbours. But instead of pointers of type Node* they are simply integers that correspond to indices in the nodes **vector**.

This makes the rest of the exercise much easier to implement. (But check the course on graph for a comparative discussion on how to represent a graph).

For example, you have a **Node** object named **myNode**, of which you know that it has **one** neighbour. And for this neighbour you want to find out the **id**. Then it would look like this:

```
// get the neighbour index stored in the first element of the neighbor vector attribute of myNode:
int neighbourIndex = myNode.getNeighbours()[0];
```

```
// from the neighbor index you get the neighbor node itself from the vector nodes. You can call its id
// getter method by using the operator . on the vector element.
string neighbourId = nodes[neighbourIndex].getId();
```

Your first task is to implement 2 methods:

- **addEdge** has to check that the index parameter (named **to** ) is not already in the Node list of neighbours. If it is the case it displays an error message in the terminal and returns from the function. Otherwise it adds the index to its vector of neighbours.
- **deleteEdge** has to check that the index parameter named **to** ) is already in the Node list of neighbor. If it is the case it removes the index from its vector of neighbours. Do not use the the **erase** method ; use the alternative presented in the course.

```
#include <iostream>
#include <vector>
#include <assert.h>
#include <queue>

using namespace std;

class Node{
private:
    const string id;
    vector<int> neighbours;
public:
    Node(const string& id):id(id){};
    string getId() const{return id;}

    void addEdge(const int& to);

    void deleteEdge(const int& to);

    const vector<int>& getNeighbours() const{
        return neighbours;
    }
};
```

## **ex b.2)** (niveau 1)

Now we would like to have a function which adds nodes and edges to the graph. Here's the code, with additional info below (important info, we introduce you to a quite useful tool : the `assert()` function):

```
// permanent vector for storing Nodes
static vector<Node> nodes;

void addNode(const string& id){
    // ToDo
}

void addEdge(const string& idA, const string& idB){
    // ToDo
}
```

As you can see, the functions are not a member of the class Node , they are placed next to the vector of Nodes. In both cases, you are given a string **id** that identifies a node.

- **addNode** : you have to make sure that the **id** is unique. There must be no existing node in the vector nodes with the same **id** already.
- **addEdge**: you need to make sure that the two **id**s are valid, e.g. that nodes with these ids exist in the vector nodes.

If these conditions fail (id is not unique, or id does not exist), the program should **terminate** with an error message. How can you do that easily?

> A simple and convenient way is the **assert** function. This function asks for a boolean expression as parameter. If it evaluates to **true** nothing happens, but if it evaluates to **false** the **assert** function terminates your program and prints the boolean expression in the terminal. Here is a basic example :

```
#include <assert.h>

int main(){
      assert(false && "custom error message");
      return 0;
}
```

Try the code above and look at the output. Re-run the code after changing **false** by **true**. Then use **assert** for the implementation of the two functions in this exercise ; just replace "false" by a Boolean expression expressing the expected "**normal**" context ; if the expression is false, **assert** will terminate the program.

# ex b.3) (niveau 1)

Implement a **<<** operator for the class **Node**. It should print the **id**, the **number** of neighbours and the **id** for each neighbour. Here is its declaration :

```
ostream& operator<<(ostream& os, const Node& node);
```

### check your code with some scaffolding code:

Here's a main that executes your code. Make sure everything works:

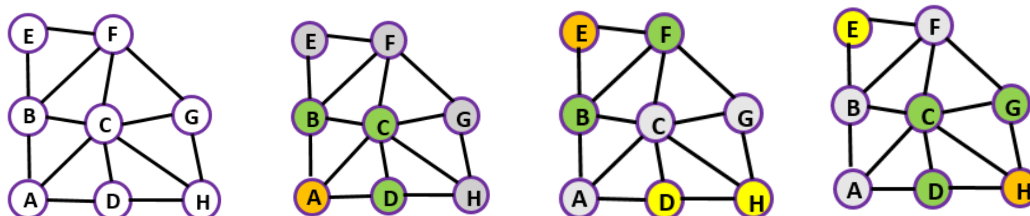
```
int main(){
    // this code should work after ex b.2)
    addNode("1");
    addNode("2");
    addNode("3");
    addNode("4");
    addNode("5");
    addNode("6");
    addNode("7");

    addEdge("1", "2");
    addEdge("1", "3");
    addEdge("2", "4");
    addEdge("2", "5");
    addEdge("5", "6");

    // after ex b.3)
    cout<<nodes[0];
```

# ex b.4) (niveau 2)

Implement a ***breadthFirst graph traversal*** that prints out all the **ids** of nodes visited, starting from the index of a given starting node. Here is an example of graph (left) and the resulting ***breadthFirst*** traversal starting from **A**, then from **E** and finally from **H**. The **start** node is in orange, the first generation of traversed nodes is in green, the second in grey and the third in yellow if any.



Let's examine the traversal starting from **A**. The ***breadthFirst*** traversal first enters the 3 neighbour nodes of A in a queue = the green generation, then it retrieves one node at a time, in the *first-in first-out* order. For each extracted node, it adds its neighbours to the queue but o*nly if they are not already there*, this will build the grey generation. The traversal stops when the queue is empty.

Here's the function signature:

```
void breadthFirst(const int& startingIndex){
    // ToDo
}
```

To implement this, you can use the C++ **queue** datastructure. Here's a few tips on how to use it:

```
#include <iostream>
#include <queue>

using namespace std;

int main(){

   // declaring one queue
   queue<int> myQueue;

   // entering three ints in the queue
   myQueue.push(1);
   myQueue.push(2);
   myQueue.push(3);

   // read (but NOT remove) the item at the front of the queue
   cout << myQueue.front()<< endl;

   // remove (only, void type) the item at the front of the queue
   // to get the item before it is removed, you need to use front()
   myQueue.pop();

   // the method empty() is true if the queue is empty, else false
   cout << myQueue.empty() << endl;

   return 0;
}
```

Note: the C++ **queue** offers only an interface for handling a *first-in first-out* queue. In particular it does not allow you to examine its content with **[ ]**. However the ***breadthFirst*** traversal requires to check the content of the queue to prevent adding an element that is already inside the queue. We propose that you adopt this solution to this design issue:

- create an additional **vector** that will accumulate the elements that have been found and put into the **queue**. You can access to any element from a **vector** and make the checks you need.
- The **queue** is emptied one by one in a loop ; the traversal ends when there is nothing left.
- Conversely the **vector** accumulates the *found* node indices until it contains them all.

## **ex b.5)** (niveau 1)

Congrats, you made it 😊
This last exercise is just for fun: You could try and reproduce the graph that was used in the lecture to explain Dijkstra's algorithm.

Note that the weights associated to the edges in that graph are not taken into account in the breadthFirst traversal. That’s what makes it different from the Dijkstra algorithm.