C++ Cheat Sheet

You can find below information that can help with the implementation of your project. Some of them might be familiar from your Object Oriented Programming course as well.

1. Structure

In C++ code is divided into two types of files: header files (with the extension .h) and code files (with the extension .cpp). In the .h file we have the declarations while the actual implementation goes in the .cpp file.

Normally, for the ADT implementation your project should contain the following files:

- **Container.h** (obviously, it is not going to be called container, you will use the name of the exact container you have to implement, for example Bag.h, Map.h, etc.)
    - This is where you have the declaration of your container class, together with its attributes and methods.
- **Container.cpp**
    - This is where you have the implementation of the operations enumerated in the container class declaration.
- **Iterator.h** (if you have ADT PriorityQueue or Matrix, you do not have an Iterator)
    - This is where you have the declaration of your iterator class, together with its attributes and methods
- **Iterator.cpp**
    - This is where you have the implementation of the operations enumerated in the iterator class declaration.

2. Includes

The next section is valid only for those projects, where there is an iterator class as well.

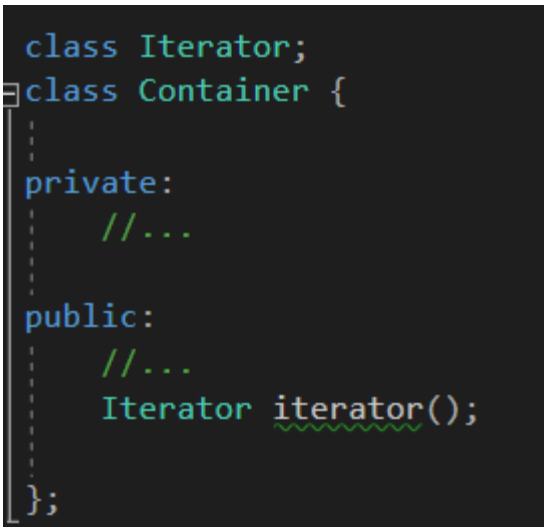
Normally, Container.h should have an include for Iterator.h, because your container class will have an operation called *iterator* which returns an element of type Iterator.

However, Iterator.h should also include Container.h, because you will keep a variable of type container as an attribute in the Iterator.

If you have both includes, it will lead to a circular dependency and a lot of compile time errors. The solution for this problem is the following:

You will include the Container.h file in the Iterator.h, but you will NOT include the Iterator.h file in the Container.h file. This leads to a compile error in Container.h, where the compiler complains that it does not know the Iterator type returned by the *iterator* operation. To avoid this, you should add

a *forward declaration* to class Iterator. This means that before the container class, you will simply write: *class Iterator;* without any further implementation for the class (obviously it will be implemented in the Iterator.h file). This forward declaration is enough for C++ to know that there will be a class called Iterator.



```
class Iterator;
class Container {

private:
    //...

public:
    //...
    Iterator iterator();

};
```

Obs.: in the .cpp files however you can include everything circular dependencies are not a problem in those files.

3. **Private/Public**

   The representation (attributes) of your container and iterator MUST be private. It must be hidden. The operations of both classes are public, but only those operations that exist in the interface of the ADT (the operations you are going to put in the project stage). You are allowed to add other operations in the container and/or iterator, but they must be private. They can be helper functions used for implementing the other ones, but they are not allowed to be visible from outside of the class. More precisely, you are not allowed to have getters and setters for your attributes. Representation is hidden (private) and this means that you are not allowed to return them!

4. **Friend classes**

   The representation of your container and iterator will be private. But the iterator needs access to the representation of the container, in order to access the elements. But you are not allowed to make the representation public. The solution for such situations is to use the *friend class* mechanism.
   If in your container class you add

```
friend class Iterator;
```

This means that the Iterator class will have access to everything declared private in the Container class.

5. Pair

If your project is to implement a Map, a MultiMap, a SortedMap or a SortedMultimap the *getCurrent* operation from the iterator will have to return a key-value pair. However, in C++ a function cannot return two things, so you need to group the key and value together somehow. One simple way to do this is to use the *pair* provided by C++. As its name suggests, it is simply a pair, a way of putting two things together. You can specify the type of the elements from the pair by using templates: `pair<string, int>` means a pair made of a string and an int. In order to use pair you need to include *utility.* An example of how to define and use a pair:

```
pair<string, int> mypair;
mypair.first = "abc";
mypair.second = 11;

pair<string, int> mypair2 = pair<string, int>("abc", 11);

string firstElem = mypair2.first;
int secondElem = mypair.second;
```

Obs. You might need something similar of you have PriorityQueue as well, to keep together an element and its priority.

6. Typedef

If you use pair, or if you have operations in your ADT which return a NULL_TVALUE or some special values in different situations, it might be a good idea to use typedefs and defines to make your implementation easier to understand and to change.

For example, if you need to use `pair<string, int>` (discussed at the previous point) often, it might be useful to assign it a new, shorter, simpler name, for example:

```
typedef pair<string, int> TPair;
```

This means that you have defined a new type, called TPair, which is another name for a pair made of a string and an int. The previous piece of code can now be simplified as:

```
TPair mypair;
mypair.first = "abc";
mypair.second = 11;

TPair mypair2 = TPair("abc", 11);

string firstElem = mypair2.first;
int secondElem = mypair.second;
```

Similarly you can define constant values using the define instruction. For example, if you have ADT Map, search should return the value associated to a key. But if that key is not in the Map, search

should return NULL_TVALUE. What NULL_TVALUE actually is, depends on your application (and on the type of the values). It could be the number 0 or -1, it could be an empty string, etc. Your code will be more readable if, instead of hardcoding this special value in your implementation, you just define it:

```
#define NULL_TVALUE -1
```

Now you can use NULL_TVALUE in your implementation