Software Design and Implementation Seminar B

This exercise implements a dynamic array class `DynArray;`

It will be an alternative to std::vector though the more sophisticated versions may have some extras. The Professional version (*aka* "Vanilla") should implement the class as declared below and in the VS project; Premium and Ultimate have extra functionality that is described but part of the exercise is for you to derive the interface (function/operator prototypes)

Pro functionality	possible prototype	comments
constructor(s)	`DynArray();`	default version – can allocate (default) buffer if desired, or wait for first insert
	`DynArray(DynArray);`	copy constructor
	`DynArray(int);`	can specify initial size, individual elements must be default-constructed
destructor	`~DynArray();`	avoid memory leaks – return any new-ed memory
container queries	`int size() const;`	returns the number of elements in the DynArray
	`int capacity() const;`	returns the maximum size available without re-allocation of the internal buffer
	`bool empty();`	returns bool indicating if there are any elements (size != 0)
element addition/deletion	`void push_back(ComponentType);`	adds the parameter to the back of the DynArray; buffer grows (by factor of ~1.5) if already full
	`void pop_back();`	removes the last element from the DynArray
element lookup	`ComponentType back();`	returns the last element
	`ComponentType front();`	returns the first element
	`ComponentType get(int);`	returns a specific element
element overwrite	`void set(ComponentType, int);`	overwrites a specific element (param) with a new value (param1)
clear	`void zap();`	clear all elements

```
class ComponentType;   //forward declaration –
                       //probably replaced a typedef

class DynArray         //Level 1 version (Professional)
{
private:
      //your choice (data & helper functions)
public:
      DynArray();
      DynArray(DynArray);
      ~DynArray();
      DynArray(int);
      int size() const;
      int capacity() const;
      bool empty();
      void push_back(ComponentType);
      void pop_back();
      ComponentType back();
      ComponentType front();
      ComponentType get(int);
      void set(ComponentType, int);
      void zap();
};
```

The definition above is not set in stone – the parameter and return types could be improved and the constructor prototypes(s) need some thought. Use every opportunity to identify const functions as such (there are more than 2).

ComponentType could be a class as suggested by the forward declaration, but it is far more likely to be a typedef; a template would be a more sophisticated choice, but is not necessary.

Out-of-range access must be policed; assertions are the most straightforward approach.

Premium functionality	comments
constructor(s)	• that fill any constructed slots with a specific (parameter?) value • reserve space (capacity set, but no inserts)
individual element removal & insertion	• expensive operation, but useful functionality – specify by index?
assignment	• assign a deep copy of the DynArray to another instance

equality	• predicate (bool function) to determine if 2 DynArray are equal
	  ○ are they same size AND
	  ○ do the elements match (value & order)

shrink	• return memory when capacity > size by copying to smaller buffer

Ultimate functionality	comments
operator[ ]	• for element lookup • for element assignment
append (not by repeated push_back)	• `.append(const DynArray);` • operator+=
dequeue behaviour	• insert / remove at front of DynArray
templates	• generalise ComponentType using templates
output	`cout << myDynArray;` would output <first, second, …penultimate, last>
more generally, submissions at this level should be aware of more subtle issues • are objects copied from the container actually copied • are objects that deleted from the container destructed	