Map ADT requirements

A map should be an associative container whose elements are ordered based on the unique key value they have been assigned. These keys are unique, and allow for the data associated with the key to be looked up and accessed quickly. The maps size changes to fit the number of elements contained within it.

template <typename tKey, typename tMValue> class CMyMap

{

public:

//the datatype being stored in the map

struct SValueType

{

tKey smKey;

tMValue smMappedValue;

};

//Comparison function

typedef bool(\*fpComparrisonFunction)(tKey, tKey);

fpComparrisonFunction fpCompare;//function to compare two keytypes. returns true if first value is higher, false otherwise

private:

// - Iterrators - //

SValueType\* mpFirst;//pointer to the first element in the map

SValueType\* mpLast;//pointer to the last element in the map

SValueType\* mpItter;//pointer to the valuetypes held in the map. used to loop through the mapped values

// - other values - //

int mSize;//the size of the map

public:

CMyMap(fpComparrisonFunction compFunc);

~CMyMap();

// - Getters - //

bool IsEmpty();//returns true if the map is empty

int GetSize();//returns the number of elements in the map

//returns a pointer to the first element of the map

SValueType\* Begin();

//returns a pointer to the last element of the map

SValueType\* End();

//returns a pointer to the mapped value associated with the given key,

//returns nullptr if no such key exists

tMValue\* GetByKeyValue(tKey key);

//returns a pointer to the mapped value at the given index in the map,

//returns nullptr if index is out of range of the map

tMValue\* GetByIndexValue(int i);

// - Overloaded Operators - //

tMValue\* operator[](int index);

void operator=(const CMyMap &aMap);

// - Interface Functions - //

//inserts a new element into the map,

//if it doesn't already exists. returns true if inserted, false otherwise

bool Insert(SValueType\* newElement);

//if it doesn't already exists, it inserts it into the map.

//if it already exists, it changes the mapped value to the new mapeed value

void InsertOrAssign(SValueType\* newElement);

//creates and inserts a new element if the new key doesnt already exist in the map

//returns true if a new element is created and inserted, false otherwise

bool Emplace(tKey Key, tMValue mapedValue);

//creates and inserts a new element if the new key doesnt already exist in the map

//If the new key already exists, updates the mapped value to the newly offered mapped value

void EmplaceOrAssign(tKey Key, tMValue mapedValue);

//removes an element with the given key from the map

//returns true if the element was found and removed, false otherwise

bool EraseByKey(tKey Key);

//removes and deletes all the elements in the current map

void Clear();

//Print contents of the map

void PrintMap();

};

Map explanation of implementation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Function | Input | Expected Output | Actual Output | Result |
| 1 | IsEmpty | N/A | true | true | pass |
| 2 | GetSize | N/A | 0 | 0 | pass |
| 3 | GetByKeyValue | 1 | a | a | pass |
| 4 | GetByIndexValue | 0 | a | a | pass |
| 5 | EmplaceOrAssign | 1,z | z | z | pass |
| 6 | IsEmpty | N/A | false | false | pass |
| 7 | GetSize | N/A | 10 | 10 | pass |
| 8 | Begin | N/A | Key Value of 1 | Key Value of 1 | pass |
| 9 | End | N/A | Key Value of 10 | Key Value of 10 | pass |
| 10 | [] Operator | 0 | z | z | pass |
| 11 | EraseByKey | 1 | index 0 to be 'b' | index 0 to be 'b' | pass |
| 12 | Clear | N/A | Empty Map | Empty Map | pass |
| 13 | = operator | N/A | All Mapped values equal | All Mapped values equal | pass |

|  |  |
| --- | --- |
| Aspect | Explanation of implementation |
| Operation of key methods | **GetByKeyValue**- uses a linear search to compare all the keys in the map to the key provided. It will return a pointer to the value mapped at the key if it finds one, and throws an exception if the key is not found within the map. The exception is a string “key not found”  **GetByIndexValue**- if the given index is within the range of the map, it returns a pointer to the mapped value at the given index. If the index is out of the maps range, it throws an exception “provided index is outside of the maps range”  **Insert**- Checks to see if the key of the new element is already in the map, and if it finds the key value already exists it returns false, ending the function without inserting anything. also it uses the comparison function to determine where in the map it should go, by checking to see at what index number key values are greater than the new elements key value. If the key does not already exist, it creates a new array that is one bigger than the old array, and populates it with the values of the old array, plus the new element. It then calls the clear method to delete the old array, and assigns the maps pointers to point at the new array. Also increases the size of the map.  **InsertOrAssign**- does the same thing as Insert, except if it finds the key already exists, instead of ending the function it changes the mapped value at that key to the mapped value of the new element. If the key does not already exist, then it functions identically to Insert.  **Emplace**- constructs a new element after being provided with a key and a mapped value, and then calls the insert method, passing this newly constructed element. Returns false if the element was not inserted  **EmplaceOrAssign**- Does the same as emplace, but calls the InsertOrAssign method Instead. Returns nothing.  **EraseByKey**- Checks if the provided key already exists in the map, if it doesn’t it returns false. If the key does exist, then it creates a new array of elements, but one smaller than the current array, and populates it with the contents of the current array, apart from the element whose key match the one provided. Then calls the clear method on the original array to delete it, and has the map point at the new array.  **Clear**- Deletes the array currently pointed at by the map. Then sets the size to zero and makes all the iterators point at nullptr.  **Printmap**- prints the contents of the map, keys then mapped value. |
| Appropriate operators | **=Operator**- calls the Clear method to delete the current map contents, then creates a new array of equal size to the provided maps array, then copy’s then contents of that map to the new array. Then it sets the map to point at the new array.  **[]Operator**-calls the GetByIndex method passing the int provided. |
| Memory Safety | The Class uses the Clear method to delete the contents of the arrays it creates using new. It also sets all pointers to null when doing so, to ensure no strange behaviour arises. |
| Persistence | The class has no persistence. |
| Flexibility | The class has high levels of flexibility, as it uses templates for both the key value and the mapped value, allowing them to be any type. It also takes a pointer to a comparison function to ensure the contents are sorted properly by the key value |
| Robust (meets expectations, error prevention & error handling) | The program meets all the expectations of a map, and throws sensible exceptions in functions where common errors might occur, such as being given an index that is out of the arrays bounds. It also provides no functions that allow the user to alter the key values, only the mapped values, this prevents the user from unwittingly breaking something, such as making it so the map is no longer sorted by key value. It checks all inputs to make sure they are sensible and consistent with the requirements of a map, such as by preventing there being identical keys within the map. |
| Other relevant issues |  |

|  |  |  |
| --- | --- | --- |
| Characteristics | Mark Range | Statement of what you have achieved |
| A flexible implementation to make a robust type with most appropriate operators to make the class natural to use (e.g. no size constraints). Thorough test program comparing a range of features with STL map. No significant errors. Professional explanation of implementation of the facilities provided. | 41-49 | I believe I have achieved all of the requirements for this mark range, as my program is both flexible (in that it can hold any type of data, through the use of template) and robust, handling most common errors you might expect. My test program evaluates all of the functions clearly and outputs it to a text document, as well as comparing the speed at which my map preforms processor intensive actions such as adding large numbers of items to the array, or searching for large numbers of values in a large array. There are no significant errors with the map. |