Homework 5

Smart Pointer & Move Semantics in C++11 and C++14

Smart pointers allow for automated deletion of variables requiring dealocation. The advantage of smart pointers compared to Garbage Collector mechanisms is that you know exactly when a given memory will be deallocated. An example when a programmer must remember to deallocate memory.

```
void  v_analyze_sell_data(CDatabase  *pcDb)
{
	CSellData  *pc_datapack;
	pc_datapack = pcGetSellDataFromDb(pcDb);
	/*do sth with data*/
	delete  pc_datapack;
}//void  v_analyze_sell_data()
```

In the example above, an object of type `CSellData` is returned by a function that pulls the object from some repository. At the end of the `v_analyze_sell_data` procedure, it must be deleted, which the programmer may forget.

If we want the dynamically allocated memory to be automatically deleted when exiting the `v_analyze_sell_data` procedure, then we should "wrap" it in some object that can be statically allocated. Then the procedure may look like below:

```
void  v_analyze_sell_data(CDatabase  *pcDb)
{
	CMySmartPointer  c_dpack(pcGetSellDataFromDb(pcDb));
	/*do sth with data*/
}//void  v_analyze_sell_data()
```

The `CMySmartPointer` object receives a pointer to a `CSellData` object for storing. The `c_dpack` object will be removed from the stack when exiting the `v_analyze_sell_data` procedure. The destructor of the `c_dpack` class should delete the stored pointer to an object of the `CSellData` class. The CMySmartPointer class may look like below.

```
class CMySmartPointer
{
public:
	CMySmartPointer(CSellData  *pcPointer) { pc_pointer = pcPointer; }
	~CMySmartPointer() { delete  pc_pointer; }

	CSellData& operator*() { return(*pc_pointer); }
	CSellData* operator->() { return(pc_pointer); }

private:
	CSellData  *pc_pointer;
};//class CMySmartPointer
```

Thanks to the implementation as above, the `CSellData` object will be deleted when exiting the `v_analyze_sell_data` procedure. What's more, thanks to overloading the * and -> operators, it will be easy to refer to the stored pointer to an object of the `CSellData` class. For example as below:

```
c_dpack->vPrintData();
(*c_dpack).vPrintData();
```

Attention. Consider whether instead of a pointer to a specific type, you can used a more general pointer. If you are able indicate this type of indicator, think about the pros and cons of such a solution.

There may be a situation in which the programmer wants to have more than one intelligent pointers, which store the same pointer. To handle this situation, you must overload the copy constructor (at least). It may look like this:

```
CMySmartPointer(const CMySmartPointer  &pcOther) { pc_pointer = pcOther.pc_pointer; }
```

Attention! The above implementation leads to an error! If the copy constructor is executed as above, the destructors of the two smart pointers will try to delete the same memory. This problem can be solved by entering of appeals counter object.

```
class  CRefCounter
{
public:
	CRefCounter() { i_count; }

	int iAdd() { return(++i_count); }
	int iDec() { return(--i_count); };
	int iGet() { return(i_count); }

private:
	int  i_count;
};//class  CRefCounter
```

The appeals counter object is created in the smart pointer class.

```
class CMySmartPointer
{
public:
	CMySmartPointer(CSellData  *pcPointer)
	{
		pc_pointer = pcPointer;
		 pc_counter = new(CRefCounter);
		pc_counter->iAdd();
	}//CMySmartPointer(CSellData  *pcPointer)

	CMySmartPointer(const CMySmartPointer  &pcOther)
	{
		pc_pointer = pcOther.pc_pointer;
		pc_counter = pcOther.pc_counter;
		pc_counter->iAdd();
	}//CMySmartPointer(const CMySmartPointer  &pcOther)

	~CMySmartPointer()
	{
		if (pc_counter->iDec())
		{
			delete  pc_pointer;
			delete  pc_counter;
		}//if (pc_counter->iDec())
	}//~CMySmartPointer()

	CSellData& operator*() { return(*pc_pointer); }
	CSellData* operator->() { return(pc_pointer); }
```

```
private:
	CRefCounter  *pc_counter;
	CSellData  *pc_pointer;
};//class CMySmartPointer
```

In the above implementation, many smart pointers can store the same pointer and will have the same of appeals counter. However, it will not causes an error. The objects pointed to by `pc_counter` and `pc_pointer` will be deleted when the last intelligent pointer that stores them is deleted.

There are situations when in some functions we have a statically created object and we want to transfer the value of this object to the outside. This is a typical situation for operator overloading, for example:

```
CNumber CNumber::operator+(CNumber &cNum)
{
	CNumber  c_result;
	/*create result*/
	return(c_result);
}//CNumber CNumber::operator+(CNumber &cNum)
```

The above implementation of the add operator is convenient due to if the `c_result` object was dynamically allocated and the operator returned a pointer to `CNumber` instead of the `CNumber` value, then if the result was not assigned to any variable outside, there would be a memory leak. Moreover, if the `CNumber` class allocates a lot of memory, removing `c_result` by the operator (at the end) can be considered as a memory waste because this object will be no longer used anywhere, and yet the memory it allocated could be put to use by another object. Instead, the memory will be copied, which is time consuming.

To cope with such situations, the so-called move semantics (MS). Let's consider an example with the `CTab` class.

```
#define DEF_TAB_SIZE  10
class  CTab
{
public:
	CTab() { pi_tab = new int[DEF_TAB_SIZE]; i_size = DEF_TAB_SIZE;}
	CTab(const CTab  &cOther);
	CTab(CTab  &&cOther);
	CTab  operator=(const CTab  &cOther);
	~CTab();

	bool  bSetSize(int  iNewSize);
	int  iGetSize() { return(i_size); }
private:
	void  v_copy(const CTab  &cOther);

	int  *pi_tab;
	int  i_size;
};//class  CTab
```

Selected methods of the `CTab` class.

```
CTab::CTab(const CTab  &cOther)
{
	v_copy(cOther);
```

```
	std::cout << "Copy ";
}//CTab::CTab(const CTab  &cOther)

CTab::~CTab()
{
	if  (pi_tab != NULL)   delete  pi_tab;
	std::cout << "Destr ";
}//CTab::~CTab()

CTab  CTab::operator=(const CTab  &cOther)
{
	if (pi_tab != NULL)   delete  pi_tab;
	v_copy(cOther);

	std::cout << "op= ";

	return(*this);
}//CTab  CTab::operator=(const CTab  &cOther)

void  CTab::v_copy(const CTab  &cOther)
{
	pi_tab = new int[cOther.i_size];
	i_size = cOther.i_size;

	for (int ii = 0; ii < cOther.i_size; ii++)
		pi_tab[ii] = cOther.pi_tab[ii];
}//void  CTab::v_copy(CTab  &cOther)
```

In the traditional way (with copying) the class can be used as in the following program.

```
CTab  cCreateTab()
{
	CTab  c_result;
	c_result.bSetSize(5);
	return(c_result);
}//CTab  cCreateTab()

int i_ms_test()
{
	CTab   c_tab = cCreateTab();
	/*DO STH WITH c_tab*/
}//int i_ms_test()
```

If the compiler does not optimize the code during compilation, the `c_tab` object will be created using the copy constructor and the array will be copied. However, we can define a move constructor as below.

```
CTab::CTab(CTab  &&cOther)
{
	pi_tab = cOther.pi_tab;
	i_size = cOther.i_size;
	cOther.pi_tab = NULL;
	std::cout << "MOVE ";
}//CTab::CTab(CTab  &&cOther)
```

Instead of copying the memory, as in the copying constructor, we rewrite the pointer to the already allocated array to the new object, and set the pointer to NULL in the old object. This is important because we don't want the old object's destructor to free up array memory. The array is moved to the new object.

Declaring a move constructor alone will not cause its use. To make it happen, use the `std::move` function.

```
CTab  cCreateTab()
{
	CTab  c_result;
	c_result.bSetSize(5);
	return(std::move(c_result));
}//CTab  cCreateTab()
```

In the above example, the `c_result` object is returned by the value, but the move constructor will be used instead of the copy constructor. Note that when executing the `i_ms_test` procedure, the array pointed to by the `pi_tab` pointer of the `c_result` object will not be deleted, but passed to the c_tab object using the move constructor. At the same time, if you use the `cCreateTab` function in the such way, there will be no memory leak, because no move constructor whose argument would be the `c_result` object from the `cCreateTab` function would not be called for the `i_ignore_result` procedure. Therefore, `c_result` will delete the array when it invokes its destructor.

```
int i_ignore_result()
{
	cCreateTab();
	/*DO STH WITH */
}//int i_ignore_result()
```

REMARK
For move semantics, use the C ++ 11 standard. Other elements of the program should be made in the C ++ 98 standard.

Your tasks to do

1. Basing on the skills acquired during previous laboratories define class template that gives opportunity to create class `CMySmartPointer` as a template class.

2. Implement operator = in such way that one smart pointer can be assigned the value of other pointer. Remember that if the modified smart pointer was pointing (storing) before assigning other pointer, you should decrement the appeals counter and deallocate memory if necessary.

3. Consider what happens when the smart pointer stores the pointer to the statically allocated memory.

4. Move constructor for the CTab class is a convenient mechanism. However, it would be convenient if you could use MS, also in case as below:

```
CTab   c_tab;
CTab  c_other;
/*initialize c_tab, c_other*/
c_tab = std::move(c_other);
```

The default content of the move operator (CTab operator=(const CTab &&cOther);) is empty, so it will not meet needed expectations. Implement the move operator that is vsuitable for the CTab class.

5. Modify the CTable class made during Exercises 2 and 3. Change the operators so that they return all results by value (if they do not), but use move semantics. Check the number of copies made with and without move semantics.