Homework 4

Class Templates

Templates allow multiple uses of the same code. It is convenient when we create functions and classes that are to be used e.g. to storing some values. A typical and often used class template is vector. This class can be used to store any types, but you must specify the stored type when a vector is created. It is important that the logic of operations performed by vector is not dependent on the stored type. So it can be implemented without knowing the stored type.

In C ++, apart from class templates, you can also declare function templates, but in this homework we will focus only on classes. An example of a class template that creates an array of type T is below.

Class declaration, type T is a template type. Its definition is required at the moment when our program wants to use the CTable class (template class) generated using class template.

```
template< typename T > class CTable
{
public:
	CTable() { i_size = 0; pt_table = NULL; }
	~CTable() { if (pt_table != NULL)  delete  [] pt_table; }

	bool  bSetLength(int iNewSize);

	T*  ptGetElement(int iOffset);
	bool  bSetElement(int iOffset, T tVal);

private:
	int i_size;
	T *pt_table;
};//template< typename T > class CTable
```

In the case of template types in C ++, due to linker errors, the implementation of methods is usually placed in the header files under the class declaration. For the `CTable` class, the method definitions will be as follows.

```
template <typename T>
bool  CTable<T>::bSetLength(int iNewSize)
{
	if (iNewSize <= 0)  return(false);

	T *pt_new_table;
	pt_new_table = new T[iNewSize];

	if (pt_table != NULL)
	{
		int  i_min_len;
		if (iNewSize < i_size)
			i_min_len = iNewSize;
		else
			i_min_len = i_size;

		for (int ii = 0; ii < i_min_len; ii++)
			pt_new_table[ii] = pt_table[ii];

		delete[] pt_table;
	}//if (pt_table != NULL)
	pt_table = pt_new_table;
	return(true);
}//bool  CTable<T>::bSetLength(int iNewSize)


template <typename T>
T*  CTable<T>::ptGetElement(int iOffset)
{
	if ((0 <= iOffset) && (iOffset < i_size))  return(NULL);

	return(&(pt_table[iOffset]));
}//T*  CTable<T>::ptGetElement(int iOffset)


template <typename T>
bool  CTable<T>::bSetElement(int iOffset, T tVal)
{
	if ((0 <= iOffset) && (iOffset < i_size))  return(false);

	pt_table[iOffset] = tVal;

	return(true);
}//bool  CTable<T>::bSetElement(int iOffset, T tVal)
```

Please note that the `CTable` class works the same, regardless of whether the type `T` is `int, double`, user class, or pointer (single or multiple).
The use of the `CTable` class can be as follows.

```
int i_template_test()
{
	CTable<int>  c_tab_int;
	CTable<double *>  c_tab_double_point;

	c_tab_int.bSetLength(10);
	c_tab_int.bSetElement(1, 22);
	int  i_val = *(c_tab_int.ptGetElement(1));

	double  d_my_doub = 5;
	c_tab_double_point.bSetLength(2);
	c_tab_double_point.bSetElement(1, &d_my_doub);
	**(c_tab_double_point.ptGetElement(1)) = 5;
}//int i_template_test()
```

Creating methods dedicated to a given template type

Sometimes we would like some template types to work differently depending on the template type. Specialization of methods is obtained as follows. We add the `sGetKnownType` method to the `CTable` class.

```
template< typename T > class CTable
{
public:
	CTable() { i_size = 0; pt_table = NULL; }
	~CTable() { if (pt_table != NULL)  delete  [] pt_table; }

	bool  bSetLength(int iNewSize);

	T*  ptGetElement(int iOffset);
	bool  bSetElement(int iOffset, T tVal);
```

```
       CString  sGetKnownType();
private:
       int i_size;
       T *pt_table;
};//template< typename T > class CTable
```

The implementation of the `sGetKnownType` method looks as follows.

```
template <typename T>
CString  CTable<T>::sGetKnownType()
{
       CString  s_type = "Unknown";
       return(s_type);
}//CString  CTable<T>::sGetKnownType()
template <>
CString  CTable<int>::sGetKnownType()
{
       CString  s_type = "INT";
       return(s_type);
}//CString  CTable<int>::sGetKnownType()
```

An Example of using:

```
int i_template_test()
{
       CTable<int>  c_tab_int;
       CTable<double *>  c_tab_double_point;

       c_tab_int.bSetLength(10);
       c_tab_int.bSetElement(1, 22);
       int  i_val = *(c_tab_int.ptGetElement(1));

       double  d_my_doub = 5;
       c_tab_double_point.bSetLength(2);
       c_tab_double_point.bSetElement(1, &d_my_doub);
       **(c_tab_double_point.ptGetElement(1)) = 5;

       CString  s_type;
       s_type = c_tab_int.sGetKnownType();
       s_type = c_tab_double_point.sGetKnownType();
}//int i_template_test()
```

If the `sGetKnownType` method is used as above, then for any template type other than `int`, `sGetKnownType` returns `"Unknown"`, and for the type `int` returns `"INT"`. There can be any number of specializations.

Behavior of the compiler when template classes are encountered.

The template class should be treated as a *recipe* for making a class. For example, when compiling the above program, the compiler will state that the CTable class will be used with the two types int and double *. Based on the class template definition, the compiler will create two independent implementations. In the first of these, instead of T, the int type will be used, and in the second, the double * type will be used.

Your task to do

For the `CTreeDynamic` class that was implemented as part of the previous exercise, change the `int` type of the `i_val` variable stored by the nodes to the template type. Make the modification in such way that the `CTreeDynamic` class will be a template class.