Правительство Российской Федерации

Государственное образовательное бюджетное учреждение

высшего профессионального образования

«Научно-исследовательский университет –

Высшая школа экономики»

Факультет: МИЭМ

Направление: *Компьютерная безопасность*

Отчет по лабораторной работе №6 (Реализация структурных паттернов)

по дисциплине

«Методы программирования»

Выполнил

Студент группы СКБ151

Михалицын Пётр

Москва, 2017.

Содержание

Описание проекта

Проект заключался реализации структурного паттерна Proxy. В нашем случае он являлся реализацией умного указателя над объектом любого класса (за счет шаблонов).

Сам проект состоит из двух одной программы main.cpp и заголовочного файла smart_ptr.h

Посмотреть полностью весь проект можно в репозитории https://github.com/lo1ol/smart-poiner

Спецификация проекта

Программа main.cpp, содержит в себе одну функцию int main() и класс klass

Класс является обыкновенной структурой и используется исключительно для проверки работоспособности шаблонной составляющей умных указателей

```
struct klass{
    int num;
    char sym;
};
```

Функция int main() так же используется исключительно для реализации тестирования над получившимися умными указателями.

Заголовочный файл smart_ptr.h содержит в себе описание класса smart_ptr и определение его методов

```
template <typename BaseType>
class smart_ptr{
private:
    BaseType* obj;
    int* count;
public:
    smart_ptr();
    smart_ptr(BaseType* ptr);
    smart_ptr(smart_ptr& sptr);
    smart_ptr& operator= (smart_ptr& sptr);
    BaseType* operator-> ();
    BaseType& operator* ();
    BaseType& operator[](int n);
    ~smart_ptr();
    int get_count();
    void reset();
    void swap(smart_ptr& sptr);

private:
    void remove();
};
```

Снизу приведены методы со соответствующей им спецификацией

```
template <typename BaseType>
smart_ptr<BaseType>::smart_ptr(BaseType* ptr)
{
   /* Construction from object
    * get ptr in instance and
    * return smart pointer
    */
}


template <typename BaseType>
smart_ptr<BaseType>::smart_ptr(smart_ptr& sptr)
{
    /* Constructor from other smart pointer
     * get reference on other smart pointer
     * copy pointer on object
     * increment count of object
```

```
     * return smart pointer
     */
}

template <typename BaseType>
smart_ptr<BaseType>::smart_ptr(){
    /* Constructer by default
     * put nullptr in pointer on obj and pointer on count
     * return smart pointer
     */
}


template <typename BaseType>
smart_ptr<BaseType>& smart_ptr<BaseType>::operator= (smart_ptr& sptr){
    /* Override operator=
     * get other smart pointer
     * call remove on current smart pointer and
     * copy attributes of other smart pointer in self
     * return self
     */
}


template <typename BaseType>
BaseType& smart_ptr<BaseType>::operator* (){
    /* Override operator *
     * return *obj
     */
}


template <typename BaseType>
BaseType* smart_ptr<BaseType>::operator-> (){
    /* Override operator ->
     * return obj
     */
}


template <typename BaseType>
smart_ptr<BaseType>::~smart_ptr (){
    /* Destructor
     */
}


template <typename BaseType>
void smart_ptr<BaseType>::remove(){
    /* if object not attached return immediately
     * otherwise decrement *count
     * if count == 0 delete obj and count
     * put in there nullptr
     */
}

template <typename BaseType>
void smart_ptr<BaseType>::reset(){
    /* if object not attached return immediately
     * otherwise call remove and put in count and obj nullptr
     */
}
```

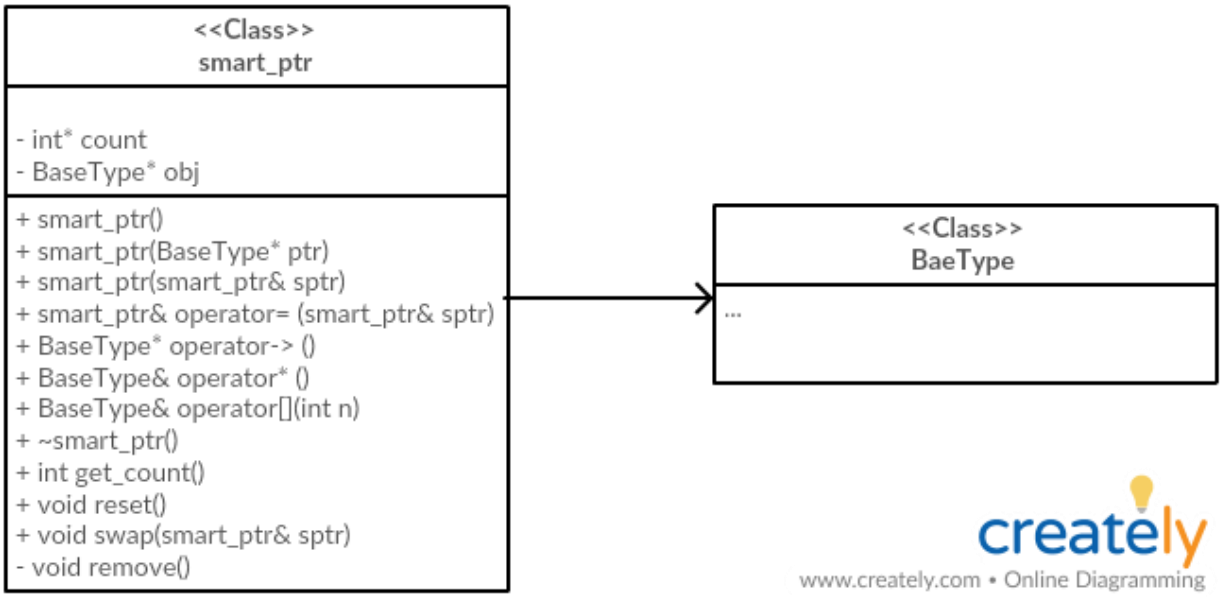
```
template <typename BaseType>
int smart_ptr<BaseType>::get_count(){
    /*return number of smart pointer that point
     * on this->obj
     */
}


template <typename BaseType>
BaseType& smart_ptr<BaseType>::operator[](int n){
    /* Override operator[]
     * get int n for operation indexing
     * return obj[n]
     */
}

template <typename BaseType>
void smart_ptr<BaseType>::swap(smart_ptr& sptr){
    /* swap two smart pointer
     * get other smart pointer
     */
}
```

Реализация структурного паттерна Proxy и его UML-диаграмма

Реализация данного паттерна оказалось весьма простой. Наш умный указатель просто хранил указатель на объект и так же подсчитывал кол-во ссылок на него. При обнулении кол-ва ссылок память на хранимый объект высвобождалась автоматически



Выводы

Паттерн Proxy является весьма хорошим решением, когда нужно скрыть некоторые действия, которые будут осуществляться автоматически в зависимости от каких-либо условий. Это поможет при разработке больших проектов и направит ваше внимание на более важные вещи, тогда как мелкие вещи будет делать за вас компьютер автоматически

Код программ и заголовочных файлов

main.cpp

```
#include <iostream>
#include <smart_ptr.h>

using namespace std;

//Some class for checking on template
struct klass{
    int num;
    char sym;
};

int main()
{
    //in this namespace must create object and delete immediately
    {
        smart_ptr<int> sptr(new int[10]);
    }
    {
        smart_ptr<int> sptr(new int[10]);
        for(int i=0; i<10; i++)
            sptr[i] = i*10;

        smart_ptr<int> sptr2;
        sptr2 = sptr;
        sptr[9] = 11;
        // check that object changes in both instance
        for(int i=0; i<10; i++)
            printf("x[%d] == %d\n", i, sptr[i]);
        printf("\n");
        //wnt to now how much instances
        std::cout<<"Number of pointer on array: "<<sptr.get_count()<<endl;
        sptr2[5] = 124;
        //remove one instaance
        sptr2.reset();
        //check that other leave
        for(int i=0; i<10; i++)
            printf("x[%d] == %d\n", i, sptr[i]);
    }
    //check in other class
    {
        smart_ptr<klass> sptr(new klass);
    }
    //make 2 different instance, swap, and check this
    {
        smart_ptr<int> sptr1(new int [10]);
        smart_ptr<int> sptr2(new int [10]);
        for(int i=0; i<10; i++)
            sptr1[i] = i;

        for(int i=0; i<10; i++)
            sptr2[i] = i*11;

        sptr1.swap(sptr2);

        cout<<"Smart pointer1\n";
        for(int i=0; i<10; i++)
            printf("x[%d] == %d\n", i, sptr1[i]);
        cout<<"Smart pointer2\n";
        for(int i=0; i<10; i++)
            printf("x[%d] == %d\n", i, sptr2[i]);
```

```
        }
    }
```

smart_ptr.h

```
#ifndef SMART_PTR_H
#define SMART_PTR_H

template <typename BaseType>
class smart_ptr{
private:
    BaseType* obj;
    int* count;
public:
    smart_ptr();
    smart_ptr(BaseType* ptr);
    smart_ptr(smart_ptr& sptr);
    smart_ptr& operator= (smart_ptr& sptr);
    BaseType* operator-> ();
    BaseType& operator* ();
    BaseType& operator[](int n);
    ~smart_ptr();
    int get_count();
    void reset();
    void swap(smart_ptr& sptr);

private:
    void remove();
};



template <typename BaseType>
smart_ptr<BaseType>::smart_ptr(BaseType* ptr)
{
    /* Construction from object
     * get ptr in instance and
     * return smart pointer
     */

        obj = ptr;
        count=new int;
        *count = 1;
        std::cout<<"catch pointer on "<<obj<<". Count = "<<*count<<std::endl;
}


template <typename BaseType>
smart_ptr<BaseType>::smart_ptr(smart_ptr& sptr)
{
    /* Constructor from other smart pointer
     * get reference on other smart pointer
     * copy pointer on object
     * increment count of object
     * return smart pointer
     */

        obj = sptr.obj;
        count = sptr.count;
        (*count)++;
        std::cout<<"catch pointer on "<<obj<<". Count = "<<*count<<std::endl;
}

template <typename BaseType>
smart_ptr<BaseType>::smart_ptr(){
    /* Constructer by default
     * put nullptr in pointer on obj and pointer on count
```

```
     * return smart pointer
     */
    obj = nullptr;
    count = nullptr;
}


template <typename BaseType>
smart_ptr<BaseType>& smart_ptr<BaseType>::operator= (smart_ptr& sptr){
    /* Override operator=
     * get other smart pointer
     * call remove on current smart pointer and
     * copy attributes of other smart pointer in self
     * return self
     */

    if (count != nullptr){
        remove();
    }
    if (this != &sptr){
        obj = sptr.obj;
        count = sptr.count;
        (*count)++;
    }
    std::cout<<"catch pointer on "<<obj<<". Count = "<<*count<<std::endl;
    return *this;
}


template <typename BaseType>
BaseType& smart_ptr<BaseType>::operator* (){
    /* Override operator *
     * return *obj
     */

    return *obj;
}


template <typename BaseType>
BaseType* smart_ptr<BaseType>::operator-> (){
    /* Override operator ->
     * return obj
     */
    return obj;
}


template <typename BaseType>
smart_ptr<BaseType>::~smart_ptr (){
    /* Destructor
     */
    remove();
}


template <typename BaseType>
void smart_ptr<BaseType>::remove(){
    /* if object not attached return immediately
     * otherwise decrement *count
     * if count == 0 delete obj and count
     * put in there nullptr
     */
```

```
    if (count == nullptr)
        return;

    (*count)--;
    if (*count == 0){
        std::cout<<"remove pointer on "<<obj<<" completele"<<std::endl;
        delete obj;
        obj = nullptr;
        delete count;
        count = nullptr;
    }
    else
        std::cout<<"remove pointer on "<<obj<<". "<<*count<<"
left"<<std::endl;
}

template <typename BaseType>
void smart_ptr<BaseType>::reset(){
    /* if object not attached return immediately
     * otherwise call remove and put in count and obj nullptr
     */
    if (count == nullptr)
        return;
    else{
        remove();
        count = nullptr;
        obj = nullptr;
    }
}

template <typename BaseType>
int smart_ptr<BaseType>::get_count(){
    /*return number of smart pointer that point
     * on this->obj
     */
    if (count == nullptr)
        return 0;
    return *count;
}


template <typename BaseType>
BaseType& smart_ptr<BaseType>::operator[](int n){
    /* Override operator[]
     * get int n for operation indexing
     * return obj[n]
     */
    return obj[n];
}

template <typename BaseType>
void smart_ptr<BaseType>::swap(smart_ptr& sptr){
    /* swap two smart pointer
     * get other smart pointer
     */
    int* tmp_count;
    BaseType* tmp_obj;
    tmp_count = sptr.count;
    tmp_obj = sptr.obj;
    sptr.count = this->count;
    sptr.obj = this->obj;
    this->count = tmp_count;
    this->obj = tmp_obj;
```

```
}

#endif // SMART_PTR_H
```