Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра: 806 «Вычислительная математика и программирование»
Факультет: «Информационные технологии и прикладная математика»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа №6.

Группа: М8О – 104Б-16
Студент: Чекушкин Денис Игоревич
Преподаватель: Поповкин Александр Викторович
Вариант: №18

Москва
2017

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

- Закрепление навыков по работе с памятью в C++.
- Создание аллокаторов памяти для динамических структур данных.

ЗАДАНИЕ

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛРNo5) спроектировать и разработать аллокатор памяти для динамической структуры данных.

Цель построения аллокатора – минимизация вызова операции malloc. Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.

Алокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-го уровня, согласно варианта задания).

Для вызова аллокатора должны быть переопределены оператор new и delete у классов-фигур.

Листинг

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

#include "allocator.h"

class Figure
{
public:
 virtual ~Figure() {} //деструктор класса
 virtual void print() const = 0;
 virtual double area() const = 0;
 bool operator > (const Figure& rhs) const;
 bool operator < (const Figure& rhs) const;
 bool operator == (const Figure& rhs) const;
 friend std::ostream& operator << (std::ostream& os, const Figure& figure);
 static Allocator allocator;
};

#endif
```

figure.cpp

```
#include "figure.h"

Allocator Figure::allocator(32, 100);
```

square.h

```
#ifndef SQUARE_H
#define SQUARE_H

#include <iostream>
#include "figure.h"

class Square : public Figure
{
public:
 Square();
 Square(std::istream& is);
 Square(size_t i);

 void print() const override;
 double area() const override;


 Square& operator = (const Square& other);
 bool operator == (const Square& other) const;
 bool operator < (const Square& rhs) const;
 bool operator > (const Square& rhs) const;

 void* operator new (unsigned int size);
 void operator delete (void* p);


 friend std::ostream& operator << (std::ostream& os, const Square& square);
 friend std::istream& operator >> (std::istream& is, Square& square);
private:
 double m_side;
};

#endif
```

sqaure.cpp

```
#include "square.h"


Square::Square()
{
 m_side = 0.0;
}
Square::Square(std::istream& is)
{
 std::cout << "================" << std::endl;
 std::cout << "Enter side: ";
 is >> m_side;
}
Square::Square(size_t i) : m_side(i) //!!
{
    std::cout << "Square passed to function. Side: " << m_side << std::endl;
}


void Square::print() const
{
 std::cout << "================" << std::endl;
 std::cout << "Figure type: square" << std::endl;
 std::cout << "Side size: " << m_side << std::endl;
}
double Square::area() const
{
 return m_side * m_side;
}
Square& Square::operator = (const Square& other)
{
 if (&other == this)
        return *this;


 m_side = other.m_side;


 return *this;
}
```

```
void* Square::operator new (unsigned int size)
{
 return Figure::allocator.allocate();
}
void Square::operator delete (void* p)
{
 Figure::allocator.deallocate(p);
}

bool Square::operator < (const Square& rhs) const
{
  return (this->area() < rhs.area());
}
bool Square::operator > (const Square& rhs) const
{
  return (this->area() > rhs.area());
}
bool Square::operator == (const Square& other) const
{
 return m_side == other.m_side;
}
bool Figure::operator > (const Figure& rhs) const
 {
   return (this->area() > rhs.area());
 }
bool Figure::operator < (const Figure& rhs) const
 {
   return (this->area() < rhs.area());
 }
bool Figure::operator == (const Figure& rhs) const
 {
   return area() == rhs.area();
 }
std::ostream& operator << (std::ostream& os, const Square& square)
{
 os << "================" << std::endl;
 os << "Figure type: square" << std::endl;
 os << "Side size: " << square.m_side << std::endl;
```

```
 return os;
}
std::istream& operator >> (std::istream& is, Square& square)
{
 std::cout << "================" << std::endl;
 std::cout << "Enter side: ";
 is >> square.m_side;
 return is;
}
std::ostream& operator << (std::ostream& os, const Figure& figure)
{
 figure.print();
 return os;
}
```

allocator.h

```
#ifndef ALLOCATOR_H
#define ALLOCATOR_H
#include <cstdlib>
#include "stack.h"

class Allocator
{
public:
 Allocator(unsigned int blockSize, unsigned int count);
 ~Allocator();

 void* allocate();
 void deallocate(void* p);
 bool hasFreeBlocks() const;
private:
size_t _size;
size_t _count;
char *_used_blocks;
void **_free_blocks;
size_t _free_count;

};
```

```
#endif
```

allocator.cpp

```
#include "allocator.h"
Allocator::Allocator(size_t size,size_t count):
_size(size),_count(count) {
 _used_blocks = (char*)malloc(_size*_count);
 _free_blocks = (void**)malloc(sizeof(void*)*_count);
 for(size_t i=0;i<_count;i++) _free_blocks[i] = _used_blocks+i*_size;
 _free_count = _count;
 std::cout << "TAllocationBlock: Memory init" << std::endl;
}
Allocator::~Allocator()
{
 if(_free_count<_count) std::cout << "TAllocationBlock: Memory leak?" <<
std::endl;
 else std::cout << "TAllocationBlock: Memory freed" <<
std::endl;
 delete _free_blocks;
 delete _used_blocks;
}
void* Allocator::allocate()
{
 void *result = nullptr;
 if(_free_count>0){
        result = _free_blocks[_free_count-1];
        _free_count--;
        std::cout << "TAllocationBlock: Allocate " << (_count-_free_count) <<
        " of " << _count << std::endl;
 } else
 {
        std::cout << "TAllocationBlock: No memory exception :-)" <<
        std::endl;
        }
return result;
}
void Allocator::deallocate(void* p)
{
 std::cout << "TAllocationBlock: Deallocate block "<< std::endl;
```

```
 _free_blocks[_free_count] = p;
 _free_count ++;
}
bool Allocator::hasFreeBlocks() const
{
 return _free_count>0;
}
```

stack.h

```
#ifndef STACK_H
#define STACK_H
#include <iostream>
#include "stack_item.h"
#include "iterator.h"

template <class T> class Stack
{
public:
 Stack();
 virtual ~Stack();
 void Push(const std::shared_ptr<T>& item);
 std::shared_ptr<T> Pop();
 unsigned int size() const;
 bool empty();
 Iterator<StackItem<T>, T> get(unsigned int index) const;
 Iterator<StackItem<T>, T> begin() const;
 Iterator<StackItem<T>, T> end() const;
 template <class K>
 friend std::ostream& operator << (std::ostream& os, const Stack<K>& stack);
private:
 std::shared_ptr<StackItem<T>> head;
 unsigned int m_size;
};
#include "stack.cpp"
#endif
```

stack.cpp

```
template <class T>
Stack<T>::Stack()
{
```

```
 head=nullptr;
}
template <class T>
Stack<T>::~Stack()
{}
template <class T>
void Stack<T>::Push(const std::shared_ptr<T>& item)
{
 std::shared_ptr<StackItem<T>> other(new StackItem<T>(item));
 other->SetNext(head);
 head=other;
}
template <class T> std::shared_ptr<T> Stack<T>::Pop() {
 std::shared_ptr<T> result;
 if (head != nullptr) {
        result = head->GetValue();
        head = head->GetNext();
 }
 return result;
}
template <class T>
unsigned int Stack<T>::size() const
{
 return m_size;
}
template <class T>
Iterator<StackItem<T>, T> Stack<T>::get(unsigned int index) const
{
 if (index >= size())
        return end();
 Iterator<StackItem<T>, T> it = begin();
 while (index > 0)
 {
        ++it;
        --index;
 }
 return it;
}
```

```
template <class T>
Iterator<StackItem<T>, T> Stack<T>::begin() const
{
 return Iterator<StackItem<T>, T>(head);
}
template <class T>
Iterator<StackItem<T>, T> Stack<T>::end() const
{
 return Iterator<StackItem<T>, T>(nullptr);
}
template <class K>
std::ostream& operator << (std::ostream& os, const Stack<K>& stack)
{
 if (stack.size() == 0)
 {
        os << "================" << std::endl;
        os << "Stack is empty" << std::endl;
 }
 else
        for (std::shared_ptr<K> item : stack)
               item->print();

 return os;
}
```

stack_item.h

```
#ifndef STACK_ITEM_H
#define STACK_ITEM_H
#include <memory>
template <class T> class StackItem
{
public:
 StackItem(const std::shared_ptr<T>& item);
 std::shared_ptr<StackItem<T>> SetNext(std::shared_ptr<StackItem>& m_next);
 std::shared_ptr<StackItem<T>> GetNext();
 std::shared_ptr<T> GetValue() const;

private:
 std::shared_ptr<T> m_item;
```

```
 std::shared_ptr<StackItem<T>> m_next;
};

#include "stack_item.cpp"

#endif
```

stack_item.cpp

```
template <class T>
StackItem<T>::StackItem(const std::shared_ptr<T>& item)
{
 m_item = item;
}

template <class T> std::shared_ptr<StackItem<T>>
StackItem<T>::SetNext(std::shared_ptr<StackItem<T>> &next) {
 std::shared_ptr<StackItem < T>> old = this->m_next;
 this->m_next = next;
 return old;
}
template <class T>
std::shared_ptr<StackItem<T>> StackItem<T>::GetNext()
{
 return this->m_next;
}
template <class T>
std::shared_ptr<T> StackItem<T>::GetValue() const
{
 return this->m_item;
}
```

Выводы: в данной лабораторной работе я закрепил навыки работы с памятью на языке C++, получил навыки содания алокаторов. Добавил алокатор и стек, в котором он должен хранить списки использованных/свободных блоков.

https://github.com/israelcode/oop/tree/master/sem2/lab6