Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»
Дисциплина «Объектно-ориентированное программирование»

Лабораторная работа №6

Тема: Основы работы с коллекциями: аллокаторы

Студент: Инютин М. А.
Группа: М8О-207Б-19
Преподаватель: Чернышев Л. Н.
Дата:
Оценка:

Москва, 2020

1. Постановка задачи

Создать шаблон динамической коллекцию, согласно варианту задания:

1. Коллекция должна быть реализована с помощью умных указателей (std::shared_ptr, std::weak_ptr). Опционально использование std::unique_ptr;
2. В качестве параметра шаблона коллекция должна принимать тип данных – фигуры;
3. Коллекция должна содержать метод доступа: pop, push, top;
4. Реализовать аллокатор, который выделяет фиксированный размер памяти (количество блоков памяти – является параметром шаблона аллокатора). Внутри аллокатор должен хранить указатель на используемый блок памяти и динамическую коллекцию указателей на свободные блоки. Динамическая коллекция должна соответствовать варианту задания (Динамический массив, Список, Стек, Очередь);
5. Коллекция должна использовать аллокатор для выделения и освобождения памяти для своих элементов;
6. Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – std::vector).
7. Реализовать программу, которая:
   - позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию;
   - позволяет удалять элемент из коллекции по номеру элемента
   - выводит на экран введённые фигуры с помощью std::for_each

Вариант 2. Квадрат, стек, список.

2. Описание программы

Будем использовать шаблонный класс квадрата и стека из предыдущей лабораторной работы. В классе стека добавим ещё один шаблонный аргумент и заменим всю работу с new и delete на работу с алокатором. Так как стек реализован с использованием std::shared_ptr, то для корректного обращения с памятью необходимо создать отдельный класс-интерфейс deleter, который нужно передать в конструктор умного указателя. Реализуем класс линейного аллокатора, который выделяет блок фиксированного размера и затем возвращает указатели на свободные блоки. Освобождать память такой аллокатор не может, поэтому метод deallocate ничего не делает. Вместо этого деструктор аллокатора освобождает весь выделенный блок памяти. Аллокатор не может вызывать конструктор для элементов, поэтому используем метод construct для инициализации динамических объектов.

3. Набор тестов

Программа принимает на вход положительное число N – количество фигур в стеке. В следующих N строках следует описание каждого квадрата: координаты левого нижнего угла и длина стороны квадрата. После этих строк программа принимает на вход индекс элемента для удаления из стека.

Тест №1

```
5
0
0 0 10
1
1 1 9
0
10 10 10
0
0 0 9
4
5 5 5
3
```

Тест №2

```
5
0
-10 -10 1
1
-5 -5 5
1
7 7 2
2
10 10 3
3
0 0 4
5
```

Тест №3
5
0
-10 -10 1
1
-5 -5 5
1
7 7 2
2
10 10 3
3
0 0 4
1

Тест №4
1
0
-1000 -1000 1000
1

4. Результат выполнения тестов

Программа выводит весь стек до и после удаления элемента.

Тест №1

Your input:
Square {(0, 0), (0, 9), (9, 9), (9, 0)}
Square {(10, 10), (10, 20), (20, 20), (20, 10)}
Square {(0, 0), (0, 10), (10, 10), (10, 0)}
Square {(1, 1), (1, 10), (10, 10), (10, 1)}
Square {(5, 5), (5, 10), (10, 10), (10, 5)}
After erase:
Square {(0, 0), (0, 9), (9, 9), (9, 0)}
Square {(10, 10), (10, 20), (20, 20), (20, 10)}
Square {(1, 1), (1, 10), (10, 10), (10, 1)}
Square {(5, 5), (5, 10), (10, 10), (10, 5)}

Тест №2

Your input:
Square {(-10, -10), (-10, -9), (-9, -9), (-9, -10)}
Square {(7, 7), (7, 9), (9, 9), (9, 7)}
Square {(10, 10), (10, 13), (13, 13), (13, 10)}
Square {(0, 0), (0, 4), (4, 4), (4, 0)}
Square {(-5, -5), (-5, 0), (0, 0), (0, -5)}
Input index to erase from stack
After erase:
Square {(-10, -10), (-10, -9), (-9, -9), (-9, -10)}
Square {(7, 7), (7, 9), (9, 9), (9, 7)}
Square {(10, 10), (10, 13), (13, 13), (13, 10)}
Square {(0, 0), (0, 4), (4, 4), (4, 0)}

Тест №3
Your input:
Square {(-10, -10), (-10, -9), (-9, -9), (-9, -10)}
Square {(7, 7), (7, 9), (9, 9), (9, 7)}
Square {(10, 10), (10, 13), (13, 13), (13, 10)}
Square {(0, 0), (0, 4), (4, 4), (4, 0)}
Square {(-5, -5), (-5, 0), (0, 0), (0, -5)}
After erase:
Square {(7, 7), (7, 9), (9, 9), (9, 7)}
Square {(10, 10), (10, 13), (13, 13), (13, 10)}
Square {(0, 0), (0, 4), (4, 4), (4, 0)}
Square {(-5, -5), (-5, 0), (0, 0), (0, -5)}

Тест №4
Your input:
Square {(-1000, -1000), (-1000, 0), (0, 0), (0, -1000)}
After erase:

5. Листинг программы

Программа разделена на файлы: allocator.hpp, square.hpp, stack.hpp, main.cpp. В первом реализация аллокатора памяти, во втором шаблонный класс квадрата, в третьем реализация шаблонной коллекции стека и итератора с использоваием аллокатора. В main.cpp взаимодействие с коллекцией.

allocator.hpp

```
#ifndef ALLOCATOR_HPP
#define ALLOCATOR_HPP

#include <list>

template<class T, std::size_t BLOCK_SIZE>
class linear_allocator_t {
private:
	std::list<T*> lst;
	T* buffer;
public:
	/*
	 * std::allocator_traits
	 */
	using allocator_type = linear_allocator_t;
	using value_type = T;
	using pointer = T*;
	using reference = T&;
	using const_reference = const T&;
	using size_type = std::size_t;

	T* allocate(const std::size_t & n) {
		if (buffer == nullptr) {
			buffer = new T[BLOCK_SIZE];
			for (std::size_t i = 0; i < BLOCK_SIZE; ++i) {
				lst.push_back(&buffer[i]);
			}
		}
		if (lst.size() < n) {
			throw(std::bad_alloc());
		} else {
			T* p = lst.front();
			for (std::size_t i = 0; i < n; ++i) {
				lst.pop_front();
			}
			return p;
		}
	}

	/*
	 * Allocator can't call constructor,
	 * so programm should call it manually
	 * OTHER_T is type of allocating variable
```

```
     * ARGS... is arguments for constructor
     */
    template<class OTHER_T, class... ARGS>
    void construct(OTHER_T* p, ARGS... arguments) {
        *p =  OTHER_T(std::forward<ARGS>(arguments)...);
    }

    /*
     * Linear allocator can't call delete
     * at the middle of allocated memory
     */
    void deallocate(pointer, std::size_t) {
        ;
    }

    linear_allocator_t() : lst(), buffer(nullptr) {
        static_assert(BLOCK_SIZE > 0);
    }

    explicit linear_allocator_t(const linear_allocator_t<T,
BLOCK_SIZE> & another_allocator) : linear_allocator_t() {
        buffer = new T[BLOCK_SIZE];
        for (std::size_t i = 0; i < BLOCK_SIZE; ++i) {
            buffer[i] = another_allocator.buffer[i];
            lst.push_back(&buffer[i]);
        }
    }

    ~linear_allocator_t() {
        delete[] buffer;
    }

    /*
     * This method is used to get
     * another allocator type
     */
    template<class OTHER_T>
    class rebind {
    public:
        using other = linear_allocator_t<OTHER_T, BLOCK_SIZE>;
    };
};

#endif /* ALLOCATOR_HPP */
```

square.hpp

```
#ifndef SQUARE_HPP
#define SQUARE_HPP

#include <iostream>
#include <tuple>

template<class T>
struct square_t {
	/* Cords of left bottom corner, side */
	std::pair<T, T> cord;
	T side;

	square_t() : cord(), side() {}
	square_t(const std::pair<T, T> & xy, T l) : cord(xy), side(l)
{}
};

template<class T>
T calc_square(const square_t<T> & Sq) {
	return Sq.side * Sq.side;
}

template<class T>
std::ostream & operator << (std::ostream & out, const square_t<T>
& sq) {
	out << "Square {";
	out << std::pair<T, T>(sq.cord.first, sq.cord.second) << ", ";
	out << std::pair<T, T>(sq.cord.first, sq.cord.second +
sq.side) << ", ";
	out << std::pair<T, T>(sq.cord.first + sq.side, sq.cord.second
+ sq.side) << ", ";
	out << std::pair<T, T>(sq.cord.first + sq.side,
sq.cord.second);
	out << "}";
	return out;
}

template<class T1, class T2>
std::ostream & operator << (std::ostream & out, const
std::pair<T1, T2> & p) {
	out << "(" << p.first << ", " << p.second << ")";
	return out;
}

#endif /* SQUARE_HPP */
```

stack.hpp

```
#ifndef STACK_HPP
#define STACK_HPP

#include <iostream>
#include <memory>

template<class T, class ALLOCATOR>
class stack_t {
private:
	struct stack_node_t;

	using allocator_type = typename ALLOCATOR::template
rebind<stack_node_t>::other;

	struct deleter {
		allocator_type stack_node_deleter;

		deleter() : stack_node_deleter() {};
		deleter(allocator_type* another_deleter) :
stack_node_deleter(another_deleter) {}

		/* std::shared_ptr uses operator() to delete memory */
		void operator() (void* ptr) {
			stack_node_deleter.deallocate((stack_node_t*)ptr,
1);
		}
	};

	struct stack_node_t {
		T data;
		std::shared_ptr<stack_node_t> next;

		stack_node_t() noexcept : data(), next(nullptr) {};
		explicit stack_node_t(const T & elem) noexcept :
data(elem), next(nullptr) {}

		friend bool operator != (const stack_node_t & lhs, const
stack_node_t & rhs) {
			return &lhs.data != &rhs.data;
		}

		friend bool operator == (const stack_node_t & lhs, const
stack_node_t & rhs) {
			return &lhs.data == &rhs.data;
		}

		friend std::ostream & operator << (std::ostream & out,
const stack_node_t & node) {
			out << node.data;
			return out;
```

```
        }
    };

public:
    class iterator {
    private:
        std::shared_ptr<stack_node_t> ptr;
    public:
        using iterator_category = std::forward_iterator_tag;
        using difference_type = std::ptrdiff_t;
        using value_type = T;
        using pointer = T*;
        using reference = T&;
        using const_reference = const T&;

        iterator() : ptr(nullptr) {}
        iterator(const std::shared_ptr<stack_node_t> &
another_ptr) : ptr(another_ptr) {}

        bool is_null() {
            return ptr == nullptr;
        }

        void unvalidate() {
            ptr = nullptr;
        }

        iterator & operator ++ () {
            if (this->ptr != nullptr) {
                this->ptr = this->ptr->next;
                return *this;
            } else {
                throw(std::runtime_error("Iterator points to
nullptr!"));
            }
        }

        bool operator != (const iterator & other_iterator) {
            return &other_iterator.ptr->data != &this->ptr-
>data;
        }

        friend std::ostream & operator << (std::ostream & out,
const iterator & it) {
            out << *(it.ptr);
            return out;
        }

        stack_node_t & operator * () {
            return *ptr;
        }
```

```
    };

private:
    std::shared_ptr<stack_node_t> top_node;
    deleter stack_deleter;

public:
    stack_t() noexcept : top_node() {};

    iterator begin() {
        return iterator(top_node);
    }

    iterator end() {
        return iterator(nullptr);
    }

    void pop() {
        if (top_node) {
            top_node = top_node->next;
        } else {
            throw(std::runtime_error("Stack is empty!"));
        }
    }

    void push(const T & elem) {
        stack_node_t* new_node =
stack_deleter.stack_node_deleter.allocate(sizeof(stack_node_t));
        stack_deleter.stack_node_deleter.construct(new_node,
elem);
        std::shared_ptr<stack_node_t> new_node_shared(new_node,
stack_deleter);
        new_node_shared->next = top_node;
        top_node = new_node_shared;
    }

    T top() {
        if (top_node) {
            return top_node->data;
        } else {
            throw(std::runtime_error("Stack is empty!"));
        }
    }

    void erase(iterator it) {
        if (it.is_null()) {
            throw(std::runtime_error("Iterator points to
nullptr!"));
        } else {
            if (*it == *top_node) {
                top_node = top_node->next;
```

```
                } else {
                    std::shared_ptr<stack_node_t> prev_node =
top_node;
                    while (*prev_node->next != *it) {
                        prev_node = prev_node->next;
                    }
                    prev_node->next = prev_node->next->next;
                    (*it).next = nullptr;
                }
                it.unvalidate();
            }
        }

        void insert(iterator it, const T & elem) {
            stack_node_t* new_node =
stack_deleter.stack_node_deleter.allocate(sizeof(stack_node_t));
            stack_deleter.stack_node_deleter.construct(new_node,
elem);
            std::shared_ptr<stack_node_t> new_node_shared(new_node,
stack_deleter);
            if (top_node) {
                if (*it == *top_node) {
                    new_node_shared->next = top_node;
                    top_node = new_node_shared;
                    it.unvalidate();
                    return;
                }
                std::shared_ptr<stack_node_t> prev_node = top_node;
                while (*prev_node->next != *it) {
                    prev_node = prev_node->next;
                }
                if (it.is_null()) {
                    prev_node->next = new_node_shared;
                } else {
                    new_node_shared->next = prev_node->next;
                    prev_node->next = new_node_shared;
                    // std::swap(prev_node->data, prev_node->next-
>data);
                }
            } else {
                top_node = new_node_shared;
            }
            it.unvalidate();
        }
};

#endif /* STACK_HPP */
```

main.cpp

```
#include <algorithm>
#include "allocator.hpp"
#include "stack.hpp"
#include "square.hpp"

/*
 * Инютин М А М8О-207Б-19
 * Создать шаблон динамической коллекцию, согласно варианту
 * задания:
 * 1. Коллекция должна быть реализована с помощью умных указателей
 * (std::shared_ptr, std::weak_ptr). Опционально использование
 * std::unique_ptr;
 * 2. В качестве параметра шаблона коллекция должна принимать тип
 * данных – фигуры;
 * 3. Коллекция должна содержать метод доступа: pop, push, top;
 * 4. Реализовать аллокатор, который выделяет фиксированный размер
 * памяти (количество блоков памяти – является параметром шаблона
 * аллокатора). Внутри аллокатор должен хранить указатель на
 * используемый блок памяти и динамическую коллекцию указателей на
 * свободные блоки. Динамическая коллекция должна соответствовать
 * варианту задания (Динамический массив, Список, Стек, Очередь).
 * 5. Коллекция должна использовать аллокатор для выделения и
 * освобождения памяти для своих элементов.
 * 6. Аллокатор должен быть совместим с контейнерами std::map и
 * std::list (опционально – vector).
 * 7. Реализовать программу, которая:
 * - позволяет вводить с клавиатуры фигуры (с типом int в качестве
 *   параметра шаблона фигуры) и добавлять в коллекцию;
 * - позволяет удалять элемент из коллекции по номеру элемента;
 * - выводит на экран введенные фигуры с помощью std::for_each;
 */

const std::size_t BLOCK_SIZE = 256;

int main() {
     auto Print = [](const auto & elem) {
          std::cout << elem << std::endl;
     };
     size_t n;
     std::cout << "Input number of squares in stack" << std::endl;
     std::cin >> n;
     int cordX, cordY, side;
     stack_t< square_t<int>, linear_allocator_t< square_t<int>,
BLOCK_SIZE > > st;
     for (size_t i = 0; i < n; ++i) {
          size_t n;
          std::cout << "Input index to insert a square" <<
std::endl;
          std::cin >> n;
```

```
            std::cout << "Input square as follows: x y a" <<
std::endl;
            std::cout << "x, y is a left bottom corner cords" <<
std::endl;
            std::cout << "a is square side" << std::endl;
            std::cin >> cordX >> cordY >> side;
            try {
                  stack_t< square_t<int>, linear_allocator_t<
square_t<int>, BLOCK_SIZE > >::iterator it = st.begin();
                  while (n--) {
                        ++it;
                  }
                  st.insert(it, square_t<int>(std::pair<int,
int>(cordX, cordY), side));
            } catch (std::runtime_error & exception) {
                  std::cout << exception.what() << std::endl;
            }
      }
      std::cout << "Your input:" << std::endl;
      std::for_each(st.begin(), st.end(), Print);
      std::cout << "Input index to erase from stack" << std::endl;
      std::cin >> n;
      try {
            stack_t< square_t<int>, linear_allocator_t<
square_t<int>, BLOCK_SIZE > >::iterator it = st.begin();
            while (n > 1) {
                  ++it;
                  --n;
            }
            st.erase(it);
      } catch (std::runtime_error & exception) {
            std::cout << exception.what() << std::endl;
      }
      std::cout << "After erase:" << std::endl;
      std::for_each(st.begin(), st.end(), Print);
      return 0;
}
```

6. Выводы

В ходе выполнения лабораторной работы я узнал про разные виды аллокаторов и изучил принципы их работы. Я реализовал свой линейный аллокатор и стек, работающий с ним. Проверил, что мой аллокатор работает со стандартными коллекциями std::set, std::vector и std::list.

Список литературы

- std::shared_ptr<T>::shared_ptr — cppreference.com
  URL: https://en.cppreference.com/w/cpp/memory/shared_ptr/shared_ptr
  (дата обращения 26.11.2020)
- std::allocator_traits — cppreference.com
  URL: https://en.cppreference.com/w/cpp/memory/allocator_traits
  (дата обращения 26.11.2020)
- Аллокаторы памяти / Хабр — Habr
  URL: https://habr.com/ru/post/505632/
  (дата обращения 26.11.2020)