Московский авиационный институт
(Национальный исследовательский университет)

Институт: «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: Савченко Илья Владимирович
Группа: 80-208
Преподаватель: Чернышов 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;
3. Коллекция должна содержать метод доступа:
   - Стек – pop, push, top;
   - Очередь – pop, push, top;
   - Список, Динамический массив – доступ к элементу по оператору [];
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 использующую аллокатор;
   - Позволяет удалять элемент из коллекции по номеру элемента;
   - Выводит на экран введенные фигуры с помощью std::for_each;

Вариант 10:

Квадрат, спискок — коллекция, список — аллокатор

2. Описание программы

square.h	
struct square<T>	Фигура “квадрат” (T - тип коорд.)
list.h	
class list<T, ALLOCATOR>	Список
struct list::item	Один элемент списка
class list::iterator	Итератор на списке

void list::Push(T);
T list::Top();
void list::Pop();
 	
void list::insert(iterator&, T);
void list::erase(iterator&);
 	
iterator list::begin();
iterator list::end()
	Основные функции линейного списка (итераторы — ЛР 5)

allocator.h	
class allocator<T, BLOCK_AMOUNT>	Аллокатор T* buffer — указатель на буфер памяти std::vector<T*> free_blocks — динамический массив, в котором хранятся указатели на свободные блоки памяти
struct allocator::rebind<U>	Шаблонная структура для изменения аллоцируемого типа данных (нужна для совместимости со стд. функциями)
T* allocate() void deallocate(T* p)	После выделения памяти, и сохранения поинтеров в free_blocks, через данные функции происходит передача поинтеров на свободные ячейки памяти во вне (а также из забор обратно)
int main()	Драйвер код

3. Набор тестов

Вводятся размеры квадратов для хранения
Программа позволяет добавлять и удалять элементы из списка как с конца, так и в любой позиции

Пример работы можно увидеть в пункте 4

4. Результаты выполнения тестов

```
p - push
o - pop
i - peep
a - insert
r - remove
l - list
x - exit

> p 3
Allocator's constructor
Allocating buffer
Allocating 0x191c08
Pushed
> p 5
Allocating 0x191be0
Pushed
> p 8
Allocating 0x191bb8
Pushed
> p 12
Allocating 0x191b90
Pushed
> l
Square size 12
Square size 8
Square size 5
Square size 3
> a 13 2
Allocating 0x191b68
Inserted
> l
Square size 12
Square size 8
Square size 13
Square size 5
```

```
Square size 3
> r 1
Deallocating 0x191be0
Erased
> l
Square size 12
Square size 13
Square size 5
Square size 3
> o
Deallocating 0x191b90
Popped
> l
Square size 13
Square size 5
Square size 3
> i
Top: Square size 13
> x
Deallocating 0x191c08
Deallocating 0x191b68
Deallocating 0x191bb8
Allocator's destructor
```

5. Листинг программы

```
///////// main.cpp
/**
 * Савченко И.В.
 * М8О-208Б-19
 * https://github.com/ShyFly46/oop_exercise_06
 *
 * Вариант 10:
 * Реализовать шаблон класса "квадрат" и шаблон
 * динамической коллекции "список".
 *
 * Реализовать аллокатор, работающий на основе
 * списка из стандартной библиотеки.
 **/

#include <iostream>
#include <algorithm>

#include "square.h"
#include "list.h"
#include "allocator.h"

void print_menu() {
    std::cout
       << "p - push\n"
       << "o - pop\n"
       << "i - peep\n"
       << "a - insert\n"
```

```
       << "r - remove\n"
       << "l - list\n"
       << "x - exit\n"
       << "\n";
}

int main() {
      list<square<int>, allocator<square<int>, 10>> s;
      print_menu();
      char cmd;
    std::cout << "> ";
      while (std::cin >> cmd) {
      switch(cmd){
       case 'p':
             {
                   try {
                         square<int> q;
                         std::cin >> q;
                         s.push(q);
                         std::cout << "Pushed\n";
                   }
                   catch (std::exception &ex) {
                         std::cout << "Not enough memory: " << ex.what() <<
'\n';
                   }
             }
             break;

      case 'o':
             {
                   try {
                         s.pop();
                         std::cout << "Popped\n";
                   }
                   catch (std::exception &ex) {
                         std::cout << ex.what() << '\n';
                   }
             }
             break;

      case 'i':
            try {
            auto t = s.top();
            std::cout << "Top: " << t << '\n';
            }
            catch (std::exception &ex) {
            std::cout << ex.what() << '\n';
            }
             break;

       case 'a':
             {
                   square<int> q;
                   std::cin >> q;
                   unsigned int pos;
                   std::cin >> pos;
                   auto iter = s.begin();
                   try {
```

```
                        if (iter == s.end() && pos != 0) {
                              throw std::runtime_error("Invalid position");
                        }
                        for (unsigned int i = 0; i < pos; ++i) {
                              ++iter;
                              if (iter == s.end() && i != pos - 1) {
                                    throw std::runtime_error("Invalid
position");
                              }
                        }
                        s.insert(iter, q);
                        std::cout << "Inserted\n";
                  }
                  catch (std::runtime_error &ex) {
                        std::cout << ex.what() << '\n';
                  }
                  catch (std::bad_alloc &ex) {
                        std::cout << "Not enough memory: " << ex.what() <<
'\n';
                  }
            }
            break;

      case 'r':
            {
                  unsigned int pos;
                  std::cin >> pos;
                  auto iter = s.begin();
                  try {
                        if (iter == s.end()) {
                              throw std::runtime_error("Invalid position");
                        }
                        for (unsigned int i = 0; i < pos; ++i) {
                              ++iter;
                              if (iter == s.end() && i != pos) {
                                    throw std::runtime_error("Invalid
position");
                              }
                        }
                        s.erase(iter);
                        std::cout << "Erased\n";
                  }
                  catch (std::exception &ex) {
                        std::cout << ex.what() << '\n';
                  }
            }
            break;

      case 'l':
            std::for_each(s.begin(), s.end(), [](square<int> q) {
            std::cout << q << '\n';
            });
            break;

      case 'x':
            return 0;

      default:
```

```
            std::cout << "Invalid cmd\n";
             break;
      }

       std::cout << "> ";
      }
}
//////// list.h
#ifndef LIST_H
#define LIST_H

#include <memory>
#include <exception>

template<class T,class ALLOCATOR>
class list {
private:

      struct item {

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

      T value;
      std::unique_ptr<item> next = nullptr;

      item() = default;

      item(T val) : value(val) {}

      item &operator=(item const &other) noexcept {
            value = other.value;
            next = std::move(other.next);
            return *this;
      }

      bool operator!=(item &other) {
            return &value != &other.value;
      }

      // singleton allocator
      static allocator_type& get_allocator() {
            static allocator_type allocator;
            return allocator;
      }

      void* operator new(size_t size) {
            return get_allocator().allocate();
      }

      void operator delete(void* p) {
            get_allocator().destroy((item*)p);
            get_allocator().deallocate((item*)p);
      }
      };

      std::unique_ptr<item> head;
```

```
public:
	class iterator {
	private:
	item* ptr;

	friend class list;

	public:

	// iterator traits
	using difference_type = ptrdiff_t;
	using value_type = T;
	using reference = T &;
	using pointer = T*;
	using iterator_category = std::forward_iterator_tag;

	iterator() {
		ptr = nullptr;
	}

	iterator(item* node) {
		ptr = node;
	}

	iterator &operator++() {
		if (ptr != nullptr) {
		ptr = ptr->next.get();
		}
		else {
		ptr = nullptr;
		}
		return *this;
	}

	reference operator*() {
		return ptr->value;
	}

	pointer operator->() {
		return &(ptr->value);
	}

	iterator& operator=(iterator const& other) {
		ptr = other.ptr;
	}

	bool operator!=(iterator &other) {
		return ptr != other.ptr;
	}

	bool operator!=(iterator &&other) {
		return ptr != other.ptr;
	}

	bool operator==(iterator &other) {
		return ptr == other.ptr;
	}
```

```
bool operator==(iterator &&other) {
      return ptr == other.ptr;
}
};

T top() {
if (head) {
      return head->value;
}
throw std::runtime_error("List is empty");
}

void pop() {
if (head) {
      std::unique_ptr<item> tmp = std::move(head);
      head = std::move(tmp->next);
} else {
      throw std::runtime_error("List is empty");
}
}

void push(T val) {
std::unique_ptr<item> new_head = std::make_unique<item>(val);
if (head) {
      new_head->next = std::move(head);
      head = std::move(new_head);
}
else {
      head = std::move(new_head);
}
}

// iterator to the top
iterator begin() const {
if (head == nullptr) {
      return nullptr;
}
return head.get();
}

// iterator to the last element
iterator end() const {
return iterator();
}

// insert to iterator's pos
void insert(iterator &it, T val) {
if (it == begin()) {
      push(val);
}
else {
      std::unique_ptr<item> new_node = std::make_unique<item>(*it);
      new_node->next = std::move(it.ptr->next);
      *it = val;
      it.ptr->next = std::move(new_node);

}
}
```

```
	// erase from iterator's pos
	void erase(iterator &it) {
	if (it == begin()) {
		pop();
	}
	else if (it.ptr->next == nullptr) {
		auto iter = begin();
		while (iter.ptr->next->next != nullptr) {
		++iter;
		}
		iter.ptr->next = nullptr;
	}
	else {
		*it = it.ptr->next->value;
		it.ptr->next = std::move(it.ptr->next->next);
	}
	}
};


#endif

////////////// allocator.h
#ifndef ALLOCATOR_H
#define ALLOCATOR_H


#include <iostream>
#include <vector>

template<class T, size_t BLOCKS_AMOUNT>
class allocator {
private:
	T* buffer;
	std::vector<T*> free_blocks;

public:

	// allocator traits
	using value_type = T;
	using pointer = T *;
	using const_pointer = const T *;
	using size_type = size_t;

	allocator() noexcept {
	std::cout << "Allocator's constructor\n";
	buffer = nullptr;
	}

	~allocator() {
	std::cout << "Allocator's destructor\n";
	free(buffer);
	}

	// allocator type conversion
	template<class U>
	struct rebind {
```

```
	using other = allocator<U, BLOCKS_AMOUNT>;
	};

	pointer allocate() {
	// at first we should allocate memory for buffer
	if (!buffer) {
		std::cout << "Allocating buffer\n";
		buffer = (pointer)malloc(BLOCKS_AMOUNT * sizeof(T));
		for (int i = 0; i < BLOCKS_AMOUNT; ++i) {
		free_blocks.push_back(buffer + i);
		}
	}

	if (free_blocks.empty()) {
		throw std::bad_alloc();
	}

	pointer p = free_blocks[free_blocks.size() - 1];
	free_blocks.pop_back();
	std::cout << "Allocating " << p << std::endl;
	return p;

	}

	void deallocate(pointer p) {
	std::cout << "Deallocating " << p << std::endl;
	free_blocks.push_back(p);
	}

	template<typename U, typename... Args>
	void construct(U* p, Args&&... args) {
	new (p) U(std::forward<Args>(args)...);
	}

	void destroy(pointer p) {
	p->~T();
	}
};


#endif

////////////// square.h
#ifndef SQUARE_H
#define SQUARE_H

#include <iostream>

template<class T>
struct square{
	std::pair<T, T> a, b, c, d;

	square(T size = 1){
	setsize(size);
	}

    void setsize(T size){
       b.first = size;
```

```
        c.second = size;
        d.first = size;
        d.second = size;
    }
};

template<class T>
std::istream &operator>>(std::istream &in, square<T> &q) {
    T size;
      in >> size;
    q.setsize(size);
    return in;
}

template<class T>
std::ostream &operator<<(std::ostream &out, square<T> &q) {
      out << "Square size "
       << q.b.first;
    return out;
}

#endif
```

Список литературы

1. Пользовательские аллокаторы в C++ [Электронный ресурс]. URL: https://habr.com/ru/post/505632/ (дата обращения 20.04.2021).