Московский авиационный институт
(Национальный исследовательский университет)

Факультет: «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: Короткевич Л. В.
Группа: 80-208
Преподаватель: Чернышов Л.Н.
Дата:
Оценка:

Москва, 2020

1. Постановка задачи. Вариант 21.

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Классы должны иметь только публичные поля. В классах не должно быть методов, только поля. Фигуры являются фигурами вращения (равнобедренными), за исключением трапеции и прямоугольника. Для хранения координат фигур необходимо использовать шаблон std::pair.

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:

o Стек – pop, push, top;

o Очередь – pop, push, top;

o 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:

o Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор;

o Позволяет удалять элемент из коллекции по номеру элемента;

o Выводит на экран введенные фигуры с помощью std::for_each;

Цель:

- Изучение основ работы с контейнерами, знакомство концепцией аллокаторов памяти;

Вариант 21:

* Фигура: *Ромб*
* Контейнер: *Очередь*
* Аллокатор: *Дин. массив*

2. Описание программы

Программа читает данные из стандартного потока ввода, пишет в стандартный поток вывода. Операции:

- add – дДобавить фигуру в конец очереди
- erase idx – удалить фигуру по индексу ixd
- print – распечатать фигуры, содержащиеся в очереди

Программа составлена из 3 файлов: Rhombus.h, Queue.h, Allocator.h, vector.h, main.cpp.

В Rhombus.h описан класс для работы с ромбом: поля – четыре точки; наличествует также конструктор от 4 точек и метод печати ромба – вывод четырех его точек.

В Queue.h описан класс, реализация структуры данных очередь. Наличествуют итераторы, необходимые для обхода оной; описаны методы Pop, Push, Top. Есть метод, позволяющий производить удаление по элемента по его индексу в очереди. Выделение памяти для очереди проивзодится с помощью аллокатора, описанного в файле Allocator.h. В нем наличествует описание всех необходимых методов, реализующих выделение памяти. Указатели на свободные блоки памяти хранятся в динамическом массиве, описанном в файле vector.h.

3. Набор тестов

Test №1:

```
add 0 1 2 2 2 1 1 0
add 1 3 1 3 1 2 1 3
add -1 1 0 2 1 1 0 0
add 1 2 3 4 5 6 7 8
add -1 2 0 4 1 2 0 0
add 0 2 1 2 0 2 2 2
print
erase 0
print
erase 1
erase 0
erase 0
```

Test №2:

```
print
add -3 6 0 12 3 6 0 0
add 0 1 2 2 2 1 1 0
add 1 2 3 4 5 6 7 8
add -2 2 0 4 2 2 0 0
```

add 0 2 1 2 0 2 2 2
add 1 3 1 3 1 2 1 3
print
erase 1
print
erase 1
erase 0
erase 0
print
Test №3:
add -1 2 0 4 1 2 0 0
add -1 1 0 2 1 1 0 0
add -3 6 0 12 3 6 0 0
erase 2
print
add -3 6 0 12 3 6 0 0
print
erase 2
print
erase 1
print
erase 0

4. Результаты выполнения тестов

```
[leo@pc LR6]$ ./main <test1
This isn't rhombus
This isn't rhombus
Rhombus successfully added
This isn't rhombus
Rhombus successfully added
This isn't rhombus
(-1, 1) (0, 2) (1, 1) (0, 0)
(-1, 2) (0, 4) (1, 2) (0, 0)
Rhombus with index 0 was successfully deleted
(-1, 2) (0, 4) (1, 2) (0, 0)
Out of range
Rhombus with index 0 was successfully deleted
Queue is empty
[leo@pc LR6]$ ./main <test2
Queue is empty
Rhombus successfully added
This isn't rhombus
This isn't rhombus
Rhombus successfully added
This isn't rhombus
This isn't rhombus
(-3, 6) (0, 12) (3, 6) (0, 0)
(-2, 2) (0, 4) (2, 2) (0, 0)
Rhombus with index 1 was successfully deleted
(-3, 6) (0, 12) (3, 6) (0, 0)
Out of range
Rhombus with index 0 was successfully deleted
```

```
Queue is empty
Queue is empty
[leo@pc LR6]$ ./main <test3
Rhombus successfully added
Rhombus successfully added
Rhombus successfully added
Rhombus with index 2 was successfully deleted
(-1, 2) (0, 4) (1, 2) (0, 0)
(-1, 1) (0, 2) (1, 1) (0, 0)
Rhombus successfully added
(-1, 2) (0, 4) (1, 2) (0, 0)
(-1, 1) (0, 2) (1, 1) (0, 0)
(-3, 6) (0, 12) (3, 6) (0, 0)
Rhombus with index 2 was successfully deleted
(-1, 2) (0, 4) (1, 2) (0, 0)
(-1, 1) (0, 2) (1, 1) (0, 0)
Rhombus with index 1 was successfully deleted
(-1, 2) (0, 4) (1, 2) (0, 0)
Rhombus with index 0 was successfully deleted
Queue is empty
```

5. Листинг программы

main.cpp

```
/*
Короткевич Л. В.
М8О-208Б-19
github.com/anxieuse/oop_exercise_06
Вариант 21:
   Фигура: ромб
   Контейнер: очередь
   Аллокатор: дин. массив
*/

#include <iostream>
#include <utility>
#include "Rhombus.h"
#include <algorithm>
#include "Queue.h"

const int BLOCK_SIZE = 10000;

double sqDist(std::pair<int, int> a, std::pair<int, int> b)
{
   return (a.first - b.first) * (a.first - b.first) + (a.second - b.second) * (a.second - b.second);
}

bool isRhombus(std::pair<int, int> coors[4])
{
   // a-x-b
   // |   |
   // w   y
   // |   |
   // d-z-c
   std::pair<int, int> w, x, y, z;
   w = coors[0], x = coors[1], y = coors[2], z = coors[3];
```

```
    double a, b, c, d;
    a = sqDist(x, w), b = sqDist(w, z), c = sqDist(z, y), d = sqDist(y, x);
    if (a == b and b == c and c == d)
        return true;
    return false;
}

void Add(TQueue<TRhombus<int>, TAllocator<TRhombus<int>, BLOCK_SIZE>> &q)
{
    std::pair<int, int> coors[4];
    for (int i = 0; i < 4; ++i)
    {
        int x, y;
        std::cin >> x >> y;
        coors[i] = std::make_pair(x, y);
    }
    if (isRhombus(coors))
    {
        q.Push(TRhombus<int>(coors[0], coors[1], coors[2], coors[3]));
        std::cout << "Rhombus successfully added\n";
    }
    else
        std::cout << "This isn't rhombus\n";
}

void Erase(TQueue<TRhombus<int>, TAllocator<TRhombus<int>, BLOCK_SIZE>> &q)
{
    int64_t idx;

    std::cin >> idx;
    if (idx >= q.Size())
    {
        if(q.Size())
        std::cout << "Out of range\n";
        else
        std::cout << "Queue is empty\n";
    }
    else
    {
        auto it = q.begin();
        std::advance(it, idx);
        q.EraseByPos(it);
        std::cout << "Rhombus with index " << idx << " was successfully deleted\n";
    }
}

void Print(TQueue<TRhombus<int>, TAllocator<TRhombus<int>, BLOCK_SIZE>> &q)
{
    if(!q.Size()) {
        std::cout << "Queue is empty\n";
        return;
    }
```

```
    std::for_each(
        q.begin(),
        q.end(),
        [](const TRhombus<int> &rmb) {
            Print(rmb);
            std::cout << "\n";
        });
}

int main()
{
    TQueue<TRhombus<int>, TAllocator<TRhombus<int>, BLOCK_SIZE>> q;
    std::string cmd;

    while (std::cin >> cmd)
    {
        if (cmd == "add")
        {
            Add(q);
        }
        else if (cmd == "erase")
        {
            Erase(q);
        }
        else if (cmd == "print")
        {
            Print(q);
        }
        else
        {
            std::cout << "Unknown command\n";
            continue;
        }
    }
    return 0;
}
```

Rhombus.h:

```
#ifndef RHOMBUS_H
#define RHOMBUS_H

#include <utility>
#include <cmath>

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

template <class T>
struct TRhombus
```

```
{
   using type = T;
   using TVertex = std::pair<T, T>;
   TVertex A, B, C, D;

   TRhombus() = default;

   TRhombus(const TVertex &v1, const TVertex &v2, const TVertex &v3, const TVertex &v4) : A(v1), B(v2),
C(v3), D(v4)
   {
   }
};

template <class T>
void Print(const TRhombus<T> &r)
{
   std::cout << r.A << " " << r.B << " " << r.C << " " << r.D;
}

#endif
```

Queue.h:

```
#pragma once

#include <iterator>
#include <memory>

template <class T>
class list
{
private:
   struct element;  // forward-declaration
   size_t size = 0; // размер списка
public:
   list() = default;

   class forward_iterator
   {
   public:
      using value_type = T;
      using reference = value_type &;
      using pointer = value_type *;
      using difference_type = std::ptrdiff_t;
      using iterator_category = std::forward_iterator_tag;

      forward_iterator(element *ptr);
      T &operator*();
      forward_iterator &operator++();
      forward_iterator operator++(int);
      bool operator==(const forward_iterator &other) const;
      bool operator!=(const forward_iterator &other) const;

   private:
```

```
        element *it_ptr;
        friend list;
    };

    forward_iterator begin();
    forward_iterator end();
    void push_back(const T &value);
    void push_front(const T &value);
    T &front();
    T &back();
    void pop_back();
    void pop_front();
    size_t length();
    bool empty();
    void delete_by_it(forward_iterator d_it);
    void delete_by_number(size_t N);
    void insert_by_it(forward_iterator ins_it, T &value);
    void insert_by_number(size_t N, T &value);
    list &operator=(const list &other);
    T &operator[](size_t index);

private:
    struct element
    {
        T value;
        std::unique_ptr<element> next_element;
        element *prev_element = nullptr;
        element(const T &value_) : value(value_) {}
        forward_iterator next();
    };

    std::unique_ptr<element> head;
    element *tail = nullptr;
};

template <class T>
typename list<T>::forward_iterator list<T>::begin()
{
    return forward_iterator(head.get());
}

template <class T>
typename list<T>::forward_iterator list<T>::end()
{
    return forward_iterator(nullptr);
}
template <class T>
size_t list<T>::length()
{
    return size;
}
template <class T>
```

```
bool list<T>::empty()
{
   return length() == 0;
}

template <class T>
void list<T>::push_back(const T &value)
{
   if (!size)
   {
      head = std::make_unique<element>(value);
      tail = head.get();
      size++;
      return;
   }

   tail->next_element = std::make_unique<element>(value);
   element *temp = tail;
   tail = tail->next_element.get();
   tail->prev_element = temp;

   size++;
}

template <class T>
void list<T>::push_front(const T &value)
{
   size++;

   std::unique_ptr<element> tmp = std::move(head);
   head = std::make_unique<element>(value);
   head->next_element = std::move(tmp);

   if (head->next_element != nullptr)
      head->next_element->prev_element = head.get();

   if (size == 1)
   {
      tail = head.get();
   }
   if (size == 2)
   {
      tail = head->next_element.get();
   }
}

template <class T>
void list<T>::pop_front()
{
   if (size == 0)
   {
      throw "error: list is empty";
```

```
    }
    if (size == 1)
    {
        head = nullptr;
        tail = nullptr;
        size--;
        return;
    }
    head = std::move(head->next_element);
    head->prev_element = nullptr;
    size--;
}

template <class T>
void list<T>::pop_back()
{
    if (size == 0)
    {
        throw "error: list is empty";
    }
    if (tail->prev_element)
    {
        element *tmp = tail->prev_element;
        tail->prev_element->next_element = nullptr;
        tail = tmp;
    }
    else
    {
        head = nullptr;
        tail = nullptr;
    }
    size--;
}

template <class T>
T &list<T>::front()
{
    if (size == 0)
    {
        throw "error: list is empty";
    }
    return head->value;
}

template <class T>
T &list<T>::back()
{
    if (size == 0)
    {
        throw "error: list is empty";
    }
    forward_iterator i = this->begin();
```

```
    while (i.it_ptr->next() != this->end())
    {
        i++;
    }
    return *i;
}

template <class T>
list<T> &list<T>::operator=(const list<T> &other)
{
    if (this == &other)
        return *this;

    size = other.size;
    head = std::move(other.head);

    return *this;
}

template <class T>
void list<T>::delete_by_it(list<T>::forward_iterator d_it)
{
    forward_iterator i = this->begin(), end = this->end();
    if (d_it == end)
        throw "error: out of range";
    if (d_it == this->begin())
    {
        this->pop_front();
        return;
    }
    if (d_it.it_ptr == tail)
    {
        this->pop_back();
        return;
    }
    if (d_it.it_ptr == nullptr)
        throw "error: out of range";
    auto temp = d_it.it_ptr->prev_element;
    std::unique_ptr<element> temp1 = std::move(d_it.it_ptr->next_element);
    d_it.it_ptr = d_it.it_ptr->prev_element;
    d_it.it_ptr->next_element = std::move(temp1);
    d_it.it_ptr->next_element->prev_element = temp;
    size--;
}

template <class T>
void list<T>::delete_by_number(size_t N)
{
    forward_iterator it = this->begin();
    for (size_t i = 0; i < N; ++i)
    {
        ++it;
```

```
    }
    this->delete_by_it(it);
}

template <class T>
void list<T>::insert_by_it(list<T>::forward_iterator ins_it, T &value)
{
    std::unique_ptr<element> tmp = std::make_unique<element>(value);
    forward_iterator i = this->begin();
    if (ins_it == this->begin())
    {
        this->push_front(value);
        return;
    }
    if (ins_it.it_ptr == nullptr)
    {
        this->push_back(value);
        return;
    }

    tmp->prev_element = ins_it.it_ptr->prev_element;
    ins_it.it_ptr->prev_element = tmp.get();
    tmp->next_element = std::move(tmp->prev_element->next_element);
    tmp->prev_element->next_element = std::move(tmp);

    size++;
}

template <class T>
void list<T>::insert_by_number(size_t N, T &value)
{
    forward_iterator it = this->begin();
    if (N > this->length() || N < 0)
        throw "error: out of range";
    else
        for (size_t i = 0; i < N; ++i)
        {
            ++it;
        }
    this->insert_by_it(it, value);
}
template <class T>
typename list<T>::forward_iterator list<T>::element::next()
{
    return forward_iterator(this->next_element.get());
}

template <class T>
list<T>::forward_iterator::forward_iterator(list<T>::element *ptr)
{
    it_ptr = ptr;
}
```

```
template <class T>
T &list<T>::forward_iterator::operator*()
{
   return this->it_ptr->value;
}
template <class T>
T &list<T>::operator[](size_t index)
{
   if (index < 0 || index >= size)
   {
      throw "error: out of range";
   }
   forward_iterator it = this->begin();
   for (size_t i = 0; i < index; i++)
   {
      it++;
   }
   return *it;
}

template <class T>
typename list<T>::forward_iterator &list<T>::forward_iterator::operator++()
{
   if (it_ptr == nullptr)
      throw "error: out of range";
   *this = it_ptr->next();
   return *this;
}

template <class T>
typename list<T>::forward_iterator list<T>::forward_iterator::operator++(int)
{
   forward_iterator old = *this;
   ++*this;
   return old;
}

template <class T>
bool list<T>::forward_iterator::operator==(const forward_iterator &other) const
{
   return it_ptr == other.it_ptr;
}

template <class T>
bool list<T>::forward_iterator::operator!=(const forward_iterator &other) const
{
   return it_ptr != other.it_ptr;
}
```

Allocator.h:

```
#ifndef ALLOCATOR_H
#define ALLOCATOR_H
```

```
#include "vector.h"
#include <cstdlib>

template <class T, size_t BLOCK_SIZE>
class TAllocator
{
public:
    using value_type = T;
    using pointer = T *;
    using const_pointer = const T *;
    using size_type = std::size_t;

    TAllocator() : Buffer_(nullptr), FreeBlocks_(0)
    {
    }

    ~TAllocator()
    {
        free(Buffer_);
    }

    template <class U>
    struct rebind
    {
        using other = TAllocator<U, BLOCK_SIZE>;
    };

    T *allocate(size_t n)
    {
        if (!Buffer_)
        {
            Buffer_ = (T *)malloc(BLOCK_SIZE * sizeof(T));
            FreeBlocks_.Resize(BLOCK_SIZE);
            FreeBlocksFill();
        }

        int i = SearchFreeSpace(n);
        FreeBlocks_.Erase(FreeBlocks_.begin() + i - n + 1, FreeBlocks_.begin() + i + 1);
        return Buffer_ + i;
    }

    void deallocate(T *ptr, size_t n)
    {
        for (int i = 0; i < n; ++i)
        {
            FreeBlocks_.PushBack(ptr + i);
        }
    }

    template <typename U, typename... Args>
    void construct(U *p, Args &&... args)
```

```
    {
        new (p) U(std::forward<Args>(args)...);
    }

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

private:
    T *Buffer_;
    TVector<T *> FreeBlocks_;

    void FreeBlocksFill()
    {
        for (int i = 0; i < BLOCK_SIZE; ++i)
        {
            FreeBlocks_[i] = Buffer_ + i;
        }
    }

    int SearchFreeSpace(size_t n)
    {
        size_t total = 0;

        int i = FreeBlocks_.Size() - 1;
        for (; i >= 0; --i)
        {
            total = 1;
            for (int j = i; j > 0 && total < n; --j)
            {
                if (FreeBlocks_[j] - FreeBlocks_[j - 1] == 1)
                    ++total;
            }
            if (total >= n)
                break;
        }

        if (total < n)
            throw std::bad_alloc();

        return i;
    }
};

#endif
```

6. Выводы

По мере выполнения данной лабораторной работы я закрепил навыки работы с шаблонами функций, классов; получил опыт написания кастомных аллокаторов. Я также описал класс для работы с очерелью,

используя умные указатели; разработал итераторы для работы с ней же. И, главное, использовал кастомный аллокатор при выделении памяти в ней.

Фрагментация и потери в производительности, связанные с использованием динамической памяти – то, что позволяет избежать написание своих аллокаторов. Проекты, где управление и распределение памяти не продумано надлежащим образом, часто страдают от случайных сбоев после длительной сессии из-за нехватки памяти (которые, кстати, практически невозможно воспроизвести) и стоят сотни часов работы программистов, пытающихся освободить память и реорганизовать её выделение.

Литература

1. Альтернативные аллокаторы памяти [Электронный Ресурс]. URL: https://habr.com/ru/post/274827/ (дата обращения – 02.12.2020)

2. Smart Pointers Queue Implementation [Электронный Ресурс]. URL: https://codereview.stackexchange.com/questions/152942/smart-pointers-queue-implementation (дата обращения – 02.12.2020)

3. CppCon 2017: Bob Steagall “How to Write a Custom Allocator” [Электронный Ресурс] URL: https://www.youtube.com/watch?v=kSWfushlvB8 (дата обращения – 02.12.2020)