МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №8

по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент Абдуль-Хади Филипп, группа М8О-207Б-21

Преподаватель Дорохов Евгений Павлович

Условие

**Задание**

Используя структуру данных, разработанную для лабораторной работы No5, 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 варианту задания).

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

**Нельзя использовать:**

• Стандартные контейнеры std.

**Программа должна позволять:**

• Вводить произвольное количество фигур и добавлять их в контейнер;

• Распечатывать содержимое контейнера;

• Удалять фигуры из контейнера.

Описание программы

Исходный код лежит в 9 файлах:

1. main.cpp: основная программа, взаимодействие с пользователем посредством ввода команд
2. figure.h: класс фигуры
3. figure.cpp: реализация класса фигуры
4. point.h: класс фигуры
5. point.cpp: реализация класса фигуры
6. square.h: класс квадрата
7. square.cpp: реализация класса квадрата
8. tvector.h: класс вектора
9. tvector.cpp: реализация вектора
10. titerator.h: класс итератора
11. tlinkedlist.h: класс связного листа
12. tlinkedlist.cpp: реализация связного листа
13. tallocator.h: класс аллокатора
14. tallocator.cpp: реализация аллокатора

Дневник отладки

|  |  |
| --- | --- |
| **Проблема** | **Исправление** |
| Неудобно собирать без make файла | Создал make файл |

Недочёты

Отсутствуют

Выводы

Аллокаторы крайне полезный и интересный инструмент, они в некоторых случаях здорово оптимизируют использование динамической памяти путем преждевременного выделения большей области памяти, хотя это и требует от разработчика немало времени. Конечно их использование не является обязательным, но мне подобные оптимизации по душе.

Исходный код ниже:

*//MAIN.CPP*

**#include <iostream>**

**#include "square.h"**

**#include "tvector.h"**

**#include "tvector.cpp"**

**using** **namespace** std;

**int** **main**()

{

cout << "Comands:" << endl;

cout << "a - add new square (a [input])" << endl;

cout << "d - erase square by index (d [idx])" << endl;

cout << "s - set square by index (s [idx] [input])" << endl;

cout << "p - print all containing squares (p)" << endl;

cout << "q - quit (q)" << endl;

**char** running = 1;

TVector<Figure> \*vect = **new** TVector<Figure>();

**char** cmd;

**while**(running)

{

cout << "> ";

cin >> cmd;

**switch**(cmd)

{

**case** 'a':

{

vect->InsertLast(shared\_ptr<Figure>(**new** Square(cin)));

**break**;

}

**case** 'd':

{

**int** di;

cin >> di;

vect->Erase(di);

**break**;

}

**case** 's':

{

**int** si;

cin >> si;

(\*vect)[si] = shared\_ptr<Figure>(**new** Square(cin));

**break**;

}

**case** 'p':

{

**for**(Figure \*elem: \*vect)

{

cout<<\*elem<<" "<<endl;

}

**break**;

}

**case** 'q':

{

running = 0;

**break**;

}

**default**:

cout << "wrong input" << endl;

}

}

**delete** vect;

}

*//POINT.H*

**#ifndef POINT\_H**

**#define POINT\_H**

**#include <iostream>**

**class** **Point** {

**public**:

Point();

Point(std::istream &is);

Point(**double** x, **double** y);

**double** x\_;

**double** y\_;

**double** **dist**(Point& other);

Point& **operator**=(**const** Point& sq);

**bool** **operator**==(**const** Point& sq);

**friend** std::istream& **operator**>>(std::istream& is, Point& p);

**friend** std::ostream& **operator**<<(std::ostream& os, Point& p);

};

**#endif *// POINT\_H***

*//POINT.CPP*

**#include "point.h"**

**#include <cmath>**

Point::Point() : x\_(0.0), y\_(0.0) {}

Point::Point(**double** x, **double** y) : x\_(x), y\_(y) {}

Point::Point(std::istream &is) {

is >> x\_ >> y\_;

}

**double** **Point::dist**(Point& other) {

**double** dx = (other.x\_ - x\_);

**double** dy = (other.y\_ - y\_);

**return** std::sqrt(dx\*dx + dy\*dy);

}

Point& Point::**operator**=(**const** Point& sq)

{

x\_ = sq.x\_;

y\_ = sq.y\_;

**return** \***this**;

}

**bool** Point::**operator**==(**const** Point& sq)

{

**return** x\_==sq.x\_ && y\_==sq.y\_;

}

std::istream& **operator**>>(std::istream& is, Point& p) {

is >> p.x\_ >> p.y\_;

**return** is;

}

std::ostream& **operator**<<(std::ostream& os, Point& p) {

os << "(" << p.x\_ << ", " << p.y\_ << ")";

**return** os;

}

*//FIGURE.H*

**#ifndef FIGURE\_H**

**#define FIGURE\_H**

**#include <iostream>**

**#include "point.h"**

**class** **Figure**

{

**public**:

**virtual** **size\_t** **VertexesNumber**() = 0;

**virtual** **double** **Area**() = 0;

**virtual** **void** **Print**(std::ostream& os) = 0;

**virtual** **void** **Read**(std::istream& is) = 0;

**double** **calcTriangleArea**(Point p1, Point p2, Point p3);

**virtual** ~Figure() {}

**friend** std::ostream& **operator**<<(std::ostream& os, Figure& p)

{

p.Print(os);

**return** os;

}

**friend** std::istream& **operator**>>(std::istream& is, Figure& p)

{

p.Read(is);

**return** is;

}

};

**#endif *// FIGURE\_H***

*//FIGURE.CPP*

**#include "figure.h"**

**#include <cmath>**

**double** **Figure::calcTriangleArea**(Point p1, Point p2, Point p3)

{

**return** abs((p1.x\_-p3.x\_)\*(p2.y\_-p3.y\_)-(p2.x\_-p3.x\_)\*(p1.y\_-p3.y\_))/2.0;

}

*//SQUARE.H*

**#ifndef SQUARE\_H**

**#define SQUARE\_H**

**#include <iostream>**

**#include "figure.h"**

**#include "tallocator.h"**

**class** **Square**: **public** Figure

{

**public**:

Square();

~Square();

Square(std::istream &is);

Square(Point pnt1, Point pnt2, Point pnt3, Point pnt4);

**size\_t** **VertexesNumber**();

**double** **Area**();

**void** **Print**(std::ostream& os);

**void** **Read**(std::istream& is);

Square& **operator**=(**const** Square& sq);

**bool** **operator**==(**const** Square& sq);

**void**\* **operator** **new**(**size\_t** size);

**void** **operator** **delete**(**void**\* ptr);

**friend** std::istream& **operator**>>(std::istream& is, Square& r);

**friend** std::ostream& **operator**<<(std::ostream& os, Square& r);

**private**:

Point p1;

Point p2;

Point p3;

Point p4;

**static** TAllocator talloc;

};

**#endif**

*//SQUARE.CPP*

**#include "square.h"**

**#include <cmath>**

Square::Square() {}

Square::~Square() {printf("destroyed\n");}

Square::Square(std::istream &is)

{

printf("created\n");

is >> \***this**;

}

TAllocator **Square::talloc**(**sizeof**(Square),10);

Square::Square(Point pnt1, Point pnt2, Point pnt3, Point pnt4)

{

p1 = pnt1;

p2 = pnt2;

p3 = pnt3;

p4 = pnt4;

}

**size\_t** **Square::VertexesNumber**()

{

**return** 4;

}

**double** **Square::Area**()

{

**return** calcTriangleArea(p1,p2,p3)+calcTriangleArea(p3,p4,p1);

}

**void** **Square::Print**(std::ostream& os)

{

os << \***this**;

}

**void** **Square::Read**(std::istream& is)

{

is >> \***this**;

}

Square& Square::**operator**=(**const** Square& sq)

{

p1=sq.p1;

p2=sq.p2;

p3=sq.p3;

p4=sq.p4;

**return** \***this**;

}

**bool** Square::**operator**==(**const** Square& sq)

{

**return** p1==sq.p1 && p2==sq.p2 && p3==sq.p3 && p4==sq.p4;

}

**void**\* Square::**operator** **new**(**size\_t** size)

{

**return** talloc.Allocate();

}

**void** Square::**operator** **delete**(**void**\* ptr)

{

talloc.Deallocate(ptr);

}

std::istream& **operator**>>(std::istream& is, Square& r) {

is >> r.p1 >> r.p2 >> r.p3 >> r.p4;

**return** is;

}

std::ostream& **operator**<<(std::ostream& os, Square& r) {

os << "Square: " << r.p1 << " " << r.p2 << " " << r.p3 << " " << r.p4;

**return** os;

}

*//TVCETOR.H*

**#ifndef TVECTOR\_H**

**#define TVECTOR\_H**

**#include <iostream>**

**#include "square.h"**

**#include "titerator.h"**

**#include <memory>**

**template**<**class** **E**>

**class** **TVector**

{

**public**:

TVector();

TVector(**const** TVector& other);

**void** **Erase**(**int** pos);

**void** **InsertLast**(**const** std::shared\_ptr<E>& elem);

**void** **RemoveLast**();

**const** std::shared\_ptr<E>& **Last**();

std::shared\_ptr<E>& **operator**[](**const** **size\_t** idx);

**bool** **Empty**();

**size\_t** **Length**();

**void** **Clear**();

TIterator<E> **begin**();

TIterator<E> **end**();

~TVector();

**private**:

**void** **resize**(**int** newsize);

std::shared\_ptr<E> \*vals;

**int** len;

**int** rLen;

};

**#endif**

*//TVECTOR.CPP*

**#include "tvector.h"**

**#include <iostream>**

**#include <cstring>**

**template**<**class** **E**>

**TVector**<E>::TVector()

{

vals = NULL;

len = 0;

rLen = 0;

}

**template**<**class** **E**>

**TVector**<E>::TVector(**const** TVector& other)

{

len = other.len;

rLen = other.rLen;

vals = (std::shared\_ptr<E>\*)malloc(**sizeof**(std::shared\_ptr<E>)\*len);

memcpy((**void**\*)vals, (**void**\*)other.vals, **sizeof**(std::shared\_ptr<E>)\*len);

}

**template**<**class** **E**>

**void** **TVector**<E>::Erase(**int** pos)

{

**if**(len == 1)

{

Clear();

**return**;

}

vals[pos] = NULL;

**for**(**int** i=pos;i<len-1;i++)

vals[i]=vals[i+1];

len--;

**if**(len==rLen>>1)

{

resize(len);

rLen=len;

}

}

**template**<**class** **E**>

**void** **TVector**<E>::InsertLast(**const** std::shared\_ptr<E>& elem)

{

**if**(rLen)

{

**if**(len>=rLen)

{

rLen<<=1;

resize(rLen);

}

}

**else**

{

rLen=1;

resize(rLen);

}

vals[len] = elem;

len++;

}

**template**<**class** **E**>

**void** **TVector**<E>::RemoveLast()

{

Erase(len-1);

}

**template**<**class** **E**>

**const** **std**::shared\_ptr<E>& TVector<E>::Last()

{

**return** vals[len-1];

}

**template**<**class** **E**>

**std**::shared\_ptr<E>& TVector<E>::**operator**[](**const** **size\_t** idx)

{

**return** vals[idx];

}

**template**<**class** **E**>

**bool** **TVector**<E>::Empty()

{

**return** len == 0;

}

**template**<**class** **E**>

**size\_t** **TVector**<E>::Length()

{

**return** len;

}

**template**<**class** **E**>

**std**::ostream& **operator**<<(std::ostream& os, TVector<E>& obj)

{

os << '[';

**for**(**size\_t** i = 0; i < obj.Length(); i++)

{

os << obj[i].get()->Area();

**if**(i != obj.Length() - 1)

os << " ";

}

os << ']';

**return** os;

}

**template**<**class** **E**>

**void** **TVector**<E>::Clear()

{

**if**(!Empty())

{

**for**(**int** i=0;i<len;i++)

vals[i]=NULL;

**delete**[] vals;

vals = NULL;

len = 0;

rLen = 0;

}

}

**template**<**class** **E**>

**void** **TVector**<E>::resize(**int** newsize)

{

std::shared\_ptr<E> \*newVals = **new** std::shared\_ptr<E>[newsize];

**for**(**int** i=0;i<len;i++)

newVals[i]=vals[i];

**delete**[] vals;

vals = newVals;

}

**template**<**class** **E**>

**TIterator**<E> **TVector**<E>::begin()

{

**return** TIterator<E>(vals,0);

}

**template**<**class** **E**>

**TIterator**<E> **TVector**<E>::end()

{

**return** TIterator<E>(vals,len);

}

**template**<**class** **E**>

**TVector**<E>::~TVector()

{

Clear();

}

*//TITERATOR.H*

**#ifndef TITERATOR\_H**

**#define TITERATOR\_H**

**#include <memory>**

**template**<**class** **E**> **class** **TIterator**

{

**public**:

TIterator<E>(std::shared\_ptr<E> \*vct, **int** ix)

{

vect = vct;

idx = ix;

}

E\* **operator**\*()

{

**return** vect[idx].get();

}

E\* **operator**->()

{

**return** vect[idx].get();

}

**void** **operator**++()

{

idx++;

}

TIterator<E> **operator**++(**int**)

{

TIterator **iter**(vect,idx);

++(\***this**);

**return** iter;

}

**bool** **operator**==(TIterator **const**& i)

{

**return** idx == i.idx;

}

**bool** **operator**!=(TIterator **const**& i)

{

**return** idx != i.idx;

}

**private**:

std::shared\_ptr<E> \*vect;

**int** idx;

};

**#endif**

*//TLINKEDLIST.CPP*

**#include "tlinkedlist.h"**

TLinkedList::TLinkedList()

{

root = NULL;

len = 0;

}

TNode\* **TLinkedList::First**()

{

**return** GetItem(0);

}

TNode\* **TLinkedList::Last**()

{

**return** GetItem(len-1);

}

**int** **TLinkedList::Find**(**void**\* ptr)

{

**int** ret = 0;

TNode \*cur = root;

**while**(cur)

{

**if**(cur->ptr==ptr)

**return** ret;

cur = cur->next;

ret++;

}

**return** -1;

}

**void** **TLinkedList::InsertFirst**(TNode \*elem)

{

Insert(elem, 0);

}

**void** **TLinkedList::InsertLast**(TNode \*elem)

{

Insert(elem, len);

}

**void** **TLinkedList::Insert**(TNode \*elem, **size\_t** position)

{

**if**(len==0)

{

root = elem;

elem->next = NULL;

}

**else** **if**(position==0)

{

elem->next = root;

root = elem;

}

**else**

{

TNode \*last = GetItem(position-1);

elem->next = last->next;

last->next = elem;

}

len++;

}

**void** **TLinkedList::RemoveFirst**()

{

Remove(0);

}

**void** **TLinkedList::RemoveLast**()

{

Remove(len-1);

}

**void** **TLinkedList::Remove**(**size\_t** position)

{

**if**(len==1)

{

root = NULL;

}

**else** **if**(position)

{

TNode \*prev = GetItem(position-1);

prev->next = prev->next->next;

}

**else**

{

root = root->next;

}

len--;

}

**void** **TLinkedList::RemoveNulls**()

{

**while**(First()->ptr==NULL)

RemoveFirst();

TNode \*cur = root;

**while**(cur->next)

{

**if**(cur->next->ptr)

{

cur = cur->next;

}

**else**

{

cur->next = cur->next->next;

}

}

}

TNode\* **TLinkedList::GetItem**(**size\_t** idx)

{

TNode \*cur = root;

**while**(idx--)

cur = cur->next;

**return** cur;

}

**size\_t** **TLinkedList::Length**()

{

**return** len;

}

std::ostream& **operator**<<(std::ostream& os, TLinkedList& list)

{

TNode \*cur = list.root;

**while**(cur)

{

os << cur->ptr << ' ';

cur = cur->next;

}

**return** os;

}

**void** **TLinkedList::Clear**()

{

root = NULL;

len = 0;

}

TLinkedList::~TLinkedList()

{

Clear();

}

*//TLINKEDLIST.H*

**#ifndef TLINKEDLIST\_H**

**#define TLINKEDLIST\_H**

**#include <iostream>**

**class** **TNode**

{

**public**:

TNode(**void** \*data)

{

ptr = data;

}

**void** \*ptr;

TNode \*next;

};

**class** **TLinkedList**

{

**public**:

TLinkedList();

TNode\* **First**();

TNode\* **Last**();

**int** **Find**(**void**\* ptr);

**void** **InsertFirst**(TNode \*elem);

**void** **InsertLast**(TNode \*elem);

**void** **Insert**(TNode \*elem, **size\_t** position);

**void** **RemoveFirst**();

**void** **RemoveLast**();

**void** **Remove**(**size\_t** position);

**void** **RemoveNulls**();

TNode\* **GetItem**(**size\_t** idx);

**size\_t** **Length**();

**friend** std::ostream& **operator**<<(std::ostream& os, TLinkedList& list);

**void** **Clear**();

~TLinkedList();

**private**:

TNode \*root;

**int** len;

};

**#endif**

*//TALLOCATOR.CPP*

**#include "tallocator.h"**

TAllocator::TAllocator(**int** size, **int** bnchSize)

{

elemSize = size;

bunchSize = bnchSize;

notused = **new** TLinkedList();

used = **new** TLinkedList();

allocated = **new** TLinkedList();

}

**void**\* **TAllocator::Allocate**()

{

**if**(notused->Length()==0)

{

**char** \*newBlock = (**char**\*)malloc(**sizeof**(TNode)+(elemSize+**sizeof**(TNode))\*bunchSize);

TNode \*newBlockNode = **new**(newBlock) TNode(newBlock);

allocated->InsertFirst(newBlockNode);

**for**(**int** i=0;i<bunchSize;i++)

{

**char**\* offset = newBlock+**sizeof**(TNode)+i\*(elemSize+**sizeof**(TNode));

TNode \*newNode = **new**(offset+elemSize) TNode(offset);

notused->InsertFirst(newNode);

}

}

TNode \*notusedElem = notused->First();

notused->RemoveFirst();

used->InsertFirst(notusedElem);

**return** notusedElem->ptr;

}

**void** **TAllocator::Deallocate**(**void**\* ptr)

{

**int** regionSize = (**sizeof**(TNode)+(elemSize+**sizeof**(TNode))\*bunchSize);

**char** \*mptr = NULL;

**int** mid = -1;

**for**(**size\_t** i=0;i<allocated->Length();i++)

{

**char**\* memptr = (**char**\*)(allocated->GetItem(i)->ptr);

**if**(ptr>=memptr && ptr<memptr+regionSize)

{

mptr = memptr;

mid = i;

**break**;

}

}

**int** usedElem = used->Find(ptr);

TNode \*usedNode = used->GetItem(usedElem);

used->Remove(usedElem);

notused->InsertFirst(usedNode);

**char** regionUsed = 0;

TNode \*cur = used->First();

**while**(cur)

{

**if**(cur->ptr>=mptr && cur->ptr<mptr+regionSize)

{

regionUsed = 1;

**break**;

}

cur=cur->next;

}

**if**(!regionUsed)

{

free(mptr);

allocated->Remove(mid);

}

}

TAllocator::~TAllocator()

{

**for**(**size\_t** i=0;i<allocated->Length();i++)

free((**char**\*)(allocated->GetItem(i)->ptr));

**delete** notused;

**delete** used;

**delete** allocated;

}

*//TALLOCATOR.H*

**#ifndef TALLOCATOR\_H**

**#define TALLOCATOR\_H**

**#include "tlinkedlist.h"**

**class** **TAllocator**

{

**public**:

TAllocator(**int** size, **int** bunchSize);

**void**\* **Allocate**();

**void** **Deallocate**(**void**\* ptr);

~TAllocator();

**private**:

**int** elemSize;

**int** bunchSize;

TLinkedList \*notused;

TLinkedList \*used;

TLinkedList \*allocated;

};

**#endif**