**Лабораторная работа №8**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Капичников Ярослав Андреевич, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

### Цель работы

Целью лабораторной работы является:

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

### Задание

Используя структуру данных, разработанную для лабораторной работы №5, 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.

**Вариант №11**

* + Фигура 1: Прямоугольник (Rectangle)
  + Структура1: Связный список
  + Структура2: Стек

**Описание программы:**

Исходный код разделён на несколько файлов:

* point.h(cpp) – описание и реализация класса точки.
* figure.h(cpp) – описание и реализация класса фигуры.
* rectangle.h(cpp) – описание и реализация класса прямоугольника (наследуется от фигуры).
* tlinkedlist.h(cpp ) - описание и реализация класса связного списка.
* tlinkedlist\_i.h(cpp ) – описание и реализация класса отдельного элемента списка.
* iterator.h – описание класса итератора.
* Tstack.h(cpp) - описание и реализация класса стек.
* Tstack\_i.h(cpp) - описание и реализация класса отдельного элемента стека.
* Allocator.h(cpp) – описание и реализация аллокатора.

**Дневник отладки**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Дата | Событие | Действие по исправлению |
| 1 |  |  |  |
|  |  |  |  |

**Вывод:**  
 Проделав данную работу, я продолжил изучение базовых понятий ооп. По сути эта лабораторная, как и предыдущая является усовершенствованием 3 лабораторной работы. На самом деле это довольно интересный метод изучения программирования, кажется, что каждая лабораторная не сильно отличается от предыдущей, однако, если сравнить первую и последнюю, то мы заметим, какой объем работы был проделан за семестр.

**Исходный код:**

**Figure.h**

|  |
| --- |
| #pragma once |
|  | #include <iostream> |
|  | #include"point.h" |
|  | using namespace std; |
|  |  |
|  | class Figure { |
|  | public: |
|  | virtual size\_t VertexesNumber() = 0; |
|  | virtual double Area() = 0; |
|  | virtual void Print(std::ostream& os) = 0; |
|  | protected: |
|  | Point a; |
|  | Point b; |
|  | Point c; |
|  | Point d; |
|  | }; |

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); |
|  | } |
|  |  |
|  | double Point::getX() |
|  | { |
|  | return x\_; |
|  | } |
|  |  |
|  | double Point::getY() |
|  | { |
|  | return y\_; |
|  | } |
|  |  |
|  | void Point::setX(double a) |
|  | { |
|  | x\_ = a; |
|  | } |
|  |  |
|  | void Point::setY(double a) |
|  | { |
|  | y\_ = a; |
|  | } |
|  |  |
|  | std::istream& operator>>(std::istream& is, Point& p) { |
|  | is >> p.x\_ >> p.y\_; |
|  | return is; |
|  | } |
|  |  |
|  | std::ostream& operator<<(std::ostream& os,const Point& p) { |
|  | os << "(" << p.x\_ << ", " << p.y\_ << ")"; |
|  | return os; |
|  | } |
|  |  |
|  | bool operator== (Point& p1, Point& p2) |
|  | { |
|  | return (p1.getX() == p2.getY() && |
|  | p1.getY() == p2.getY()); |
|  | } |
|  |  |
|  | bool operator!= (Point& p1, Point& p2) |
|  | { |
|  | return !(p1 == p2); |
|  | } |

Point.h

|  |
| --- |
| #pragma once |
|  | #ifndef POINT\_H |
|  | #define POINT\_H |
|  |  |
|  | #include <iostream> |
|  |  |
|  | class Point { |
|  | public: |
|  | Point(); |
|  | Point(std::istream& is); |
|  | Point(double x, double y); |
|  |  |
|  | double dist(Point& other); |
|  | double getX(); |
|  | double getY(); |
|  | void setX(double a); |
|  | void setY(double a); |
|  |  |
|  | friend std::istream& operator>>(std::istream& is, Point& p); |
|  | friend std::ostream& operator<<(std::ostream& os,const Point& p); |
|  |  |
|  | friend bool operator== (Point& p1, Point& p2); |
|  | friend bool operator!= (Point& p1, Point& p2); |
|  |  |
|  | private: |
|  | double x\_; |
|  | double y\_; |
|  | }; |
|  |  |
|  | #endif |

Rectangle.cpp

|  |
| --- |
| #include <iostream> |
|  | #include"point.h" |
|  | #include"rectangle.h" |
|  | using namespace std; |
|  |  |
|  | Rectangle::Rectangle(Point a1, Point a2, Point a3, Point a4) { |
|  | a = a1; |
|  | b = a2; |
|  | c = a3; |
|  | d = a4; |
|  | } |
|  |  |
|  | Rectangle::Rectangle() { |
|  | a.setX(0); |
|  | a.setY(0); |
|  | b.setX(0); |
|  | b.setY(0); |
|  | c.setX(0); |
|  | c.setY(0); |
|  | d.setX(0); |
|  | d.setY(0); |
|  |  |
|  | } |
|  |  |
|  | double Rectangle::Area() { |
|  | double A = a.dist(b); |
|  | double B = b.dist(c); |
|  | return A \* B; |
|  | } |
|  |  |
|  | void Rectangle::Print(std::ostream& os) |
|  | { |
|  | std::cout << "Rectangle: " << a << " " << b << " " << c << " " << d << endl; |
|  | } |
|  |  |
|  | size\_t Rectangle::VertexesNumber() |
|  | { |
|  | return (size\_t)4; |
|  | } |
|  | Rectangle::Rectangle(std::istream& is) { |
|  |  |
|  | cin >> a >> b >> c >> d; |
|  | } |
|  |  |
|  | std::istream& operator>>(std::istream& is, Rectangle& p) { |
|  | is >> p.a >> p.b >> p.c >> p.d; |
|  | return is; |
|  | } |
|  |  |
|  | std::ostream& operator<<(std::ostream& os,const Rectangle& p) { |
|  | os << p.a << " " << p.b << " " << p.c << " " << p.d; |
|  | return os; |
|  | } |
|  |  |
|  | bool operator== (Rectangle& p1, Rectangle& p2) |
|  | { |
|  | return (p1.a == p2.a && |
|  | p1.b == p2.b && p1.c == p2.c && p1.d == p2.d); |
|  | } |
|  |  |
|  | bool operator!= (Rectangle& p1, Rectangle& p2) |
|  | { |
|  | return !(p1 == p2); |
|  | } |

Rectangle.h

|  |
| --- |
| #pragma once |
|  | #include <iostream> |
|  | #include"point.h" |
|  | #include"figure.h" |
|  | class Rectangle : Figure { |
|  | public: |
|  | double Area(); |
|  | void Print(std::ostream& os); |
|  | size\_t VertexesNumber(); |
|  | Rectangle(Point a1, Point a2, Point a3, Point a4); |
|  | Rectangle(std::istream& is); |
|  | Rectangle(); |
|  | friend std::istream& operator>>(std::istream& is, Rectangle& p); |
|  | friend std::ostream& operator<<(std::ostream& os,const Rectangle& p); |
|  |  |
|  | friend bool operator== (Rectangle& r1, Rectangle& r2); |
|  | friend bool operator!= (Rectangle& r1, Rectangle& r2); |
|  |  |
|  | private: |
|  |  |
|  | }; |

Tlinkedlist.cpp

|  |
| --- |
| #include "tlinkedlist.h" |
|  | #include "iterator.h" |
|  |  |
|  | template<typename T> |
|  | TLinkedList<T>::TLinkedList() { |
|  | len = 0; |
|  | head = nullptr; |
|  | } |
|  |  |
|  | template<typename T> |
|  | TLinkedList<T>::TLinkedList(const TLinkedList<T>& list) { |
|  | len = list.len; |
|  | if (!list.len) { |
|  | head = nullptr; |
|  | return; |
|  | } |
|  | head = make\_shared<TLinkedListItem<T>>(list.head->GetVal(), nullptr); |
|  | shared\_ptr<TLinkedListItem<T>> cur = head; |
|  | shared\_ptr<TLinkedListItem<T>> it = list.head; |
|  | for (size\_t i = 0; i < len - 1; ++i) { |
|  | it = it->GetNext(); |
|  | shared\_ptr<TLinkedListItem<T>> new\_item = make\_shared<TLinkedListItem<T>>(it->GetVal(), nullptr); |
|  | cur->SetNext(new\_item); |
|  | cur = cur->GetNext(); |
|  | } |
|  | } |
|  |  |
|  | template<typename T> |
|  | shared\_ptr<T> TLinkedList<T>::First() { |
|  | if (len == 0) { |
|  | return nullptr; |
|  | } |
|  | return head->GetVal(); |
|  | } |
|  |  |
|  | template<typename T> |
|  | shared\_ptr<T> TLinkedList<T>::Last() { |
|  | if (len == 0) { |
|  | return nullptr; |
|  | } |
|  | shared\_ptr<TLinkedListItem<T>> cur = head; |
|  | for (size\_t i = 0; i < len - 1; ++i) { |
|  | cur = cur->GetNext(); |
|  | } |
|  | return cur->GetVal(); |
|  | } |
|  |  |
|  | template<typename T> |
|  | void TLinkedList<T>::InsertFirst(shared\_ptr<T> figure) { |
|  | shared\_ptr<TLinkedListItem<T>> it = make\_shared<TLinkedListItem<T>>(figure, head); |
|  | head = it; |
|  | len++; |
|  | } |
|  |  |
|  | template<typename T> |
|  | void TLinkedList<T>::InsertLast(shared\_ptr<T> figure) { |
|  | if (len == 0) { |
|  | head = make\_shared<TLinkedListItem<T>>(figure, nullptr); |
|  | len = 1; |
|  | return; |
|  | } |
|  | shared\_ptr<TLinkedListItem<T>> cur = head; |
|  | for (size\_t i = 0; i < len - 1; ++i) { |
|  | cur = cur->GetNext(); |
|  | } |
|  | shared\_ptr<TLinkedListItem<T>> it = make\_shared<TLinkedListItem<T>>(figure, nullptr); |
|  | cur->SetNext(it); |
|  | len++; |
|  | } |
|  |  |
|  | template<typename T> |
|  | void TLinkedList<T>::Insert(shared\_ptr<T> figure, size\_t pos) { |
|  | if (pos > len || pos < 0) { |
|  | return; |
|  | } |
|  | shared\_ptr<TLinkedListItem<T>> cur = head; |
|  | shared\_ptr<TLinkedListItem<T>> prev = nullptr; |
|  | for (size\_t i = 0; i < pos; ++i) { |
|  | prev = cur; |
|  | cur = cur->GetNext(); |
|  | } |
|  | shared\_ptr<TLinkedListItem<T>> it = make\_shared<TLinkedListItem<T>>(figure, cur); |
|  | if (prev) { |
|  | prev->SetNext(it); |
|  | } |
|  | else { |
|  | head = it; |
|  | } |
|  | len++; |
|  | } |
|  |  |
|  | template<typename T> |
|  | void TLinkedList<T>::RemoveFirst() { |
|  | if (!len)return; |
|  | shared\_ptr<TLinkedListItem<T>> del = head; |
|  | head = head->GetNext(); |
|  | len--; |
|  | } |
|  |  |
|  | template<typename T> |
|  | void TLinkedList<T>::RemoveLast() { |
|  | if (!len)return; |
|  | if (len == 1) { |
|  | head = nullptr; |
|  | len = 0; |
|  | return; |
|  | } |
|  | shared\_ptr<TLinkedListItem<T>> cur = head; |
|  | for (size\_t i = 0; i < len - 2; ++i) { |
|  | cur = cur->GetNext(); |
|  | } |
|  | shared\_ptr<TLinkedListItem<T>> del = cur->GetNext(); |
|  | cur->SetNext(nullptr); |
|  | len--; |
|  | } |
|  |  |
|  | template<typename T> |
|  | void TLinkedList<T>::Remove(size\_t pos) { |
|  | if (!len)return; |
|  | if (pos < 0 || pos >= len)return; |
|  | shared\_ptr<TLinkedListItem<T>> cur = head; |
|  | shared\_ptr<TLinkedListItem<T>> prev = nullptr; |
|  | for (size\_t i = 0; i < pos; ++i) { |
|  | prev = cur; |
|  | cur = cur->GetNext(); |
|  | } |
|  | if (prev) { |
|  | prev->SetNext(cur->GetNext()); |
|  | } |
|  | else { |
|  | head = cur->GetNext(); |
|  | } |
|  | len--; |
|  | } |
|  |  |
|  | template<typename T> |
|  | shared\_ptr<T> TLinkedList<T>::GetItem(size\_t ind) { |
|  | if (ind < 0 || ind >= len)return nullptr; |
|  | shared\_ptr<TLinkedListItem<T>> cur = head; |
|  | for (size\_t i = 0; i < ind; ++i) { |
|  | cur = cur->GetNext(); |
|  | } |
|  | return cur->GetVal(); |
|  | } |
|  |  |
|  | template<typename T> |
|  | bool TLinkedList<T>::Empty() { |
|  | return len == 0; |
|  | } |
|  |  |
|  | template<typename T> |
|  | size\_t TLinkedList<T>::Length() { |
|  | return len; |
|  | } |
|  |  |
|  | template<typename T> |
|  | std::ostream& operator<<(std::ostream& os, const TLinkedList<T>& list) { |
|  | shared\_ptr<TLinkedListItem<T>> cur = list.head; |
|  | os << "List: \n"; |
|  | for (size\_t i = 0; i < list.len; ++i) { |
|  | os << \*cur; |
|  | cur = cur->GetNext(); |
|  | } |
|  | return os; |
|  | } |
|  |  |
|  | template<typename T> |
|  | void TLinkedList<T>::Clear() { |
|  | while (!(this->Empty())) { |
|  | this->RemoveFirst(); |
|  | } |
|  | } |
|  |  |
|  | template<typename T> |
|  | TLinkedList<T>::~TLinkedList() { |
|  | while (!(this->Empty())) { |
|  | this->RemoveFirst(); |
|  | } |
|  |  |
|  | } |
|  |  |
|  | template |
|  | class TLinkedList<Rectangle>; |
|  |  |
|  | template std::ostream& operator<<(std::ostream& os, const TLinkedList<Rectangle>& list); |
|  |  |
|  | template<typename T> |
|  | Iter<TLinkedListItem<T>, T> TLinkedList<T>::begin() { |
|  | return Iter<TLinkedListItem<T>, T>(head); |
|  | } |
|  |  |
|  | template<typename T> |
|  | Iter<TLinkedListItem<T>, T> TLinkedList<T>::end() { |
|  | Iter<TLinkedListItem<T>, T> it = begin(); |
|  | for (size\_t i = 0; i < len; ++i) { |
|  | it++; |
|  | } |
|  | return it; |
|  | } |

Tlinkedlist.h

|  |
| --- |
| #pragma once |
|  | #include "tlinkedlist\_i.h" |
|  | #include "iterator.h" |
|  |  |
|  |  |
|  | template<typename T> |
|  | class TLinkedList { |
|  | private: |
|  | size\_t len; |
|  | shared\_ptr<TLinkedListItem<T>> head; |
|  | public: |
|  | TLinkedList(); |
|  |  |
|  | TLinkedList(const TLinkedList<T>& list); |
|  |  |
|  | shared\_ptr<T> First(); |
|  |  |
|  | shared\_ptr<T> Last(); |
|  |  |
|  | void InsertFirst(shared\_ptr<T> rectangle); |
|  |  |
|  | void InsertLast(shared\_ptr<T> rectangle); |
|  |  |
|  | void Insert(shared\_ptr<T> rectangle, size\_t pos); |
|  |  |
|  | void RemoveFirst(); |
|  |  |
|  | void RemoveLast(); |
|  |  |
|  | void Remove(size\_t pos); |
|  |  |
|  | shared\_ptr<T> GetItem(size\_t ind); |
|  |  |
|  | bool Empty(); |
|  |  |
|  | size\_t Length(); |
|  |  |
|  | template<typename X> |
|  | friend std::ostream& operator<<(std::ostream& os, const TLinkedList<X>& list); |
|  |  |
|  | void Clear(); |
|  |  |
|  | virtual ~TLinkedList(); |
|  |  |
|  | Iter<TLinkedListItem<T>, T> begin(); |
|  |  |
|  | Iter<TLinkedListItem<T>, T> end(); |
|  | }; |

Tlinkedlist\_i.cpp

|  |
| --- |
| #include "tlinkedlist\_i.h" |
|  |  |
|  |  |
|  |  |
|  | template<typename T> |
|  | TLinkedListItem<T>::TLinkedListItem(shared\_ptr<T> figure, shared\_ptr<TLinkedListItem<T>> nxt) { |
|  | val = figure; |
|  | next = nxt; |
|  | } |
|  |  |
|  | template<typename T> |
|  | shared\_ptr<TLinkedListItem<T>> TLinkedListItem<T>::GetNext() { |
|  | return next; |
|  | } |
|  |  |
|  | template<typename T> |
|  | void TLinkedListItem<T>::SetNext(shared\_ptr<TLinkedListItem<T>> nxt) { |
|  | next = nxt; |
|  | } |
|  |  |
|  | template<typename T> |
|  | shared\_ptr<T> TLinkedListItem<T>::GetVal() { |
|  | return val; |
|  | } |
|  |  |
|  | template<typename T> |
|  | std::ostream& operator<<(std::ostream& os, const TLinkedListItem<T>& item) { |
|  | os << "[" << \*item.val << "] "; |
|  | return os; |
|  | } |
|  |  |
|  | template class TLinkedListItem<Rectangle>; |
|  | template std::ostream& operator<<(std::ostream& os, const TLinkedListItem<Rectangle>& item); |
|  |  |
|  | template<typename T> |
|  | TLinkedListItem<T>::~TLinkedListItem() { |
|  |  |
|  | } |
|  |  |
|  | template class TLinkedListItem<Rectangle>; |
|  | template std::ostream& operator<<(std::ostream& os, const TLinkedListItem<Rectangle>& item); |

Tlinkedlist­\_i.h

|  |
| --- |
| #pragma once |
|  |  |
|  |  |
|  | #include "rectangle.h" |
|  | #include "iostream" |
|  | #include "memory" |
|  |  |
|  | using std::shared\_ptr; |
|  | using std::make\_shared; |
|  |  |
|  | template <typename T> |
|  | class TLinkedListItem { |
|  | private: |
|  | shared\_ptr<T> val; |
|  | shared\_ptr<TLinkedListItem<T>> next; |
|  | public: |
|  | TLinkedListItem(shared\_ptr<T> rectangle, shared\_ptr<TLinkedListItem<T>> nxt); |
|  |  |
|  | void SetNext(shared\_ptr<TLinkedListItem<T>> nxt); |
|  |  |
|  | shared\_ptr<TLinkedListItem<T>> GetNext(); |
|  |  |
|  | shared\_ptr<T> GetVal(); |
|  |  |
|  | template<typename T1> |
|  | friend std::ostream& operator<<(std::ostream& os, const TLinkedListItem<T1>& item); |
|  |  |
|  | virtual ~TLinkedListItem(); |
|  | }; |

Iterator.h

|  |
| --- |
| #pragma once |
|  | #include "iostream" |
|  | #include "memory" |
|  |  |
|  | using std::shared\_ptr; |
|  |  |
|  | template<typename node, typename T> |
|  | class Iter { |
|  | public: |
|  | Iter(shared\_ptr<node> t) { |
|  | ptr = t; |
|  | } |
|  |  |
|  | shared\_ptr<T> operator\*() { |
|  | return ptr->GetVal(); |
|  | } |
|  |  |
|  | shared\_ptr<T> operator->() { |
|  | return ptr->GetVal(); |
|  | } |
|  |  |
|  | Iter<node, T> operator++() { |
|  | return ptr = ptr->GetNext(); |
|  | } |
|  |  |
|  | Iter<node, T> operator++(int) { |
|  | Iter iter(\*this); |
|  | ++(\*this); |
|  | return iter; |
|  | } |
|  |  |
|  | bool operator==(Iter<node, T> const& t) { |
|  | return ptr == t.ptr; |
|  | } |
|  |  |
|  | bool operator!=(Iter<node, T> const& t) { |
|  | return !(\*this == t); |
|  | } |
|  |  |
|  | private: |
|  | shared\_ptr<node> ptr; |
|  | }; |

Tstack.ccp

|  |
| --- |
| #include <iostream> |
|  | #include <memory> |
|  | #include "tstack.h" |
|  |  |
|  | template <class T> |
|  | TStack<T>::TStack() |
|  | { |
|  | head = nullptr; |
|  | count = 0; |
|  | } |
|  |  |
|  | template <class T> |
|  | void TStack<T>::Push(const T& item) |
|  | { |
|  | TStack\_i<T>\* tmp = new TStack\_i<T>(item, head); |
|  | head = tmp; |
|  | ++count; |
|  | } |
|  |  |
|  | template <class T> |
|  | bool TStack<T>::IsEmpty() const |
|  | { |
|  | return !count; |
|  | } |
|  |  |
|  | template <class T> |
|  | uint32\_t TStack<T>::GetSize() const |
|  | { |
|  | return count; |
|  | } |
|  |  |
|  | template <class T> |
|  | void TStack<T>::Pop() |
|  | { |
|  | if (head) { |
|  | TStack\_i<T>\* tmp = &head->GetNext(); |
|  | delete head; |
|  | head = tmp; |
|  | --count; |
|  | } |
|  | } |
|  |  |
|  | template <class T> |
|  | T& TStack<T>::Top() |
|  | { |
|  | return head->Pop(); |
|  | } |
|  |  |
|  | template <class T> |
|  | TStack<T>::~TStack() |
|  | { |
|  | for (TStack\_i<T>\* tmp = head, \*tmp2; tmp; tmp = tmp2) { |
|  | tmp2 = &tmp->GetNext(); |
|  | delete tmp; |
|  | } |
|  | } |
|  |  |
|  | template class |
|  | TStack<void\*>; |

Tstack.h

|  |
| --- |
| #pragma once |
|  | #include <iostream> |
|  | #include <memory> |
|  | #include "tstack\_i.h" |
|  |  |
|  | template <class T> |
|  | class TStack |
|  | { |
|  | public: |
|  | TStack(); |
|  | virtual ~TStack(); |
|  | void Push(const T& item); |
|  | void Pop(); |
|  | T& Top(); |
|  | bool IsEmpty() const; |
|  | uint32\_t GetSize() const; |
|  | template <class A> friend std::ostream& operator<<(std::ostream& os, const TStack<A>& stack); |
|  |  |
|  | private: |
|  | TStack\_i<T>\* head; |
|  | uint32\_t count; |
|  | }; |

Tstack\_i.ccp

|  |
| --- |
| #include <iostream> |
|  | #include <memory> |
|  | #include "tstack\_i.h" |
|  |  |
|  | template <class T> |
|  | TStack\_i<T>::TStack\_i(const T& val, TStack\_i<T>\* item) |
|  | { |
|  | value = new T(val); |
|  | next = item; |
|  | } |
|  |  |
|  | template <class T> |
|  | void TStack\_i<T>::Push(const T& val) |
|  | { |
|  | \*value = val; |
|  | } |
|  |  |
|  | template <class T> |
|  | T& TStack\_i<T>::Pop() const |
|  | { |
|  | return \*value; |
|  | } |
|  |  |
|  | template <class T> |
|  | void TStack\_i<T>::SetNext(TStack\_i<T>\* item) |
|  | { |
|  | next = item; |
|  | } |
|  |  |
|  | template <class T> |
|  | TStack\_i<T>& TStack\_i<T>::GetNext() const |
|  | { |
|  | return \*next; |
|  | } |
|  |  |
|  | template <class T> |
|  | TStack\_i<T>::~TStack\_i() |
|  | { |
|  | delete value; |
|  | } |
|  |  |
|  | template class |
|  | TStack\_i<void\*>; |

Tstack\_i.h

|  |
| --- |
| #pragma once |
|  | #include <iostream> |
|  | #include <memory> |
|  |  |
|  | template <class T> |
|  | class TStack\_i |
|  | { |
|  | public: |
|  | TStack\_i(const T& val, TStack\_i<T>\* item); |
|  | virtual ~TStack\_i(); |
|  |  |
|  | void Push(const T& val); |
|  | T& Pop() const; |
|  | void SetNext(TStack\_i<T>\* item); |
|  | TStack\_i<T>& GetNext() const; |
|  |  |
|  | private: |
|  | T\* value; |
|  | TStack\_i<T>\* next; |
|  | }; |

Allocator.h

|  |
| --- |
| #pragma once |
|  | #include <cstdlib> |
|  | #include "tstack.h" |
|  |  |
|  | class TAllocationBlock |
|  | { |
|  | public: |
|  | TAllocationBlock(size\_t size, size\_t count); |
|  | void\* allocate(); |
|  | void deallocate(void\* pointer); |
|  | bool has\_free\_blocks(); |
|  | virtual ~TAllocationBlock(); |
|  |  |
|  | private: |
|  | size\_t \_size; |
|  | size\_t \_count; |
|  | char\* \_used\_blocks; |
|  | TStack<void\*> \_free\_blocks; |
|  | size\_t \_free\_count; |
|  | }; |

Allocator.ccp

|  |
| --- |
| #include "allocator.h" |
|  | #include <iostream> |
|  |  |
|  | TAllocationBlock::TAllocationBlock(size\_t size, size\_t count) : \_size(size), \_count(count) |
|  | { |
|  | \_used\_blocks = (char\*)malloc(size \* count); |
|  | for (size\_t i = 0; i < count; i++) { |
|  | \_free\_blocks.Push(\_used\_blocks + i \* size); |
|  | } |
|  | \_free\_count = count; |
|  | std::cout << "Memory init" << "\n"; |
|  | } |
|  |  |
|  | void\* TAllocationBlock::allocate() |
|  | { |
|  | void\* result = nullptr; |
|  | if (\_free\_count == 0) { |
|  | std::cout << "No memory exception\n" << "\n"; |
|  | return result; |
|  | } |
|  | result = \_free\_blocks.Top(); |
|  | \_free\_blocks.Pop(); |
|  | --\_free\_count; |
|  | std::cout << "Allocate " << (\_count - \_free\_count) << "\n"; |
|  | return result; |
|  | } |
|  |  |
|  | void TAllocationBlock::deallocate(void\* pointer) |
|  | { |
|  | \_free\_blocks.Push(pointer); |
|  | ++\_free\_count; |
|  | std::cout << "Deallocated block\n"; |
|  | } |
|  |  |
|  | bool TAllocationBlock::has\_free\_blocks() |
|  | { |
|  | return \_free\_count > 0; |
|  | } |
|  |  |
|  | TAllocationBlock::~TAllocationBlock() |
|  | { |
|  | free(\_used\_blocks); |
|  | } |