**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 8**

Тема: Программирование классов на языке С++

Студент: Ваньков Д. А.

Группа: 80-207Б-17

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2018

Вариант задания

Вариант 22: дерево с фигурами: пятиугольник, шестиугольник,

восьмиугольник.

Контейнер 2-го уровня: стек.

Постановка задачи

### **Цель работы**

Целью лабораторной работы является:

· Знакомство с параллельным программированием в C++.

### **Задание**

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класса-контейнера .

Необходимо разработать два вида алгоритма:

· Обычный, без параллельных вызовов.

· С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

Для создания потоков использовать механизмы:

· future

· packaged\_task/async

Для обеспечения потоко-безопасности структур данных использовать:

· mutex

· lock\_guard

Нельзя использовать:

· Стандартные контейнеры std.

Программа должна позволять:

· Вводить произвольное количество фигур и добавлять их в контейнер.

· Распечатывать содержимое контейнера.

· Удалять фигуры из контейнера.

· Проводить сортировку контейнера

Описание реализации программы

Все фигуры являются подклассами виртуального класса figure, в котором описаны основные методы для фигур: *Square* — нахождение площади данной фигуры, *Print —* вывод параметров фигуры и деструктор объекта. В функции *main* стоит switch Case, на котором построен интерфейс программы. Пользователь вводит некоторое число, чтобы запустить необходимую ему функцию (подробнее в следующем пункте). Из switch Case вставка и удаление элемента, содержащего в себе фигуру, выполняется вывод всех элементов очереди и завершение приложения. Для каждой фигуры реализованы конструкторы, для вызова без параметров, для вызова с параметрами в качестве необходимого количества чисел с плавающей точкой, для вызова с потоком ввода в качестве параметра и для вызова со ссылкой на объект того же класса (конструктор копирования). В конструкторах была написана проверка введенных данных (стороны фигуры не могут быть отрицательными). Естественно в классе каждой фигуры реализованы процедуры описанные в *figure*: *Square* и *Print* и деструктор (который не отличается от стандартного деструктора из-за того, что в данных классах не использовались динамические типы)*.*  При вводе параметров конкретной фигуры, производится проверка на ее существование, в случае неверно введенных параметров – выведется сообщение об этом, и к фигуре применяются нулевые значения. Для элементов стека реализован обычный конструктор, сеттер и геттер элемента стека, геттер фигуры, оператор вывода параметров элемента стека и деструктор. Для класса стек были реализованы два конструктора, функции добавления и удаления элемента, функции получения первого и последнего элемента, оператор вывода всех элементов очереди. И наконец для удобства запуска и компиляции программы был написан *Makefile*.

Описание построения дерева

Каждому элементу присваивается свой ключ, по которому он добавляется как сын предка, ключ которого мы вводим. Если дерево пустое первый элемент встает на место корня. Если при вводе, был не найден предок с определенным ключом, элемент встает как брат для последнего сына.

Для удаления элемента из дерева нужно ввести номер его ключа. Если удаляется элемент, у которого есть дети, удаляются также все дети этого элемента.

Руководство по эксплуатации

Для того, чтобы скомпилировать программу, запустите *make* в директории с исходным кодом. Чтобы запустить программу необходимо ввести *./a.out* в командной строке. При запуске программа выведет сообщение о том, какие команды необходимо ввести для вызова конкретных функций:

1. Еще раз вывести руководство
2. Добавить пятиугольник
3. Добавить шестиугольник
4. Добавить восьмиугольник
5. Удалить фигуру из дерева и вывести её параметры
6. Вывести все дерево
7. Отсортировать
8. Вывести дерево

Для завершения программы нужно ввести любой другой символ.

Краткий обзор модулей программы:

|  |  |
| --- | --- |
| Figure.h | Модуль отвечающий за описание общего родительского класса фигур |
| Octagon.h | Заголовочный файл, описывающий класс “Восьмиугольник” |
| Octagon.cpp | Файл, отвечающий за операции над восьмиугольником, содержащий все необходимые функции и переменные, для этого класса |
| Hexagon.h | Заголовочный файл, описывающий класс “Шестиугольник” |
| Hexagon.cpp | Файл, отвечающий за операции над шестиугольником, содержащий все необходимые функции и переменные, для этого класса |
| Pentagon.h | Заголовочный файл, описывающий класс “Пятиугольник” |
| Pentagon.cpp | Файл, отвечающий за операции над пятиугольником, содержащий все необходимые функции и переменные, для этого класса |
| Tree.h | Заголовочный файл, описывающий контейнер “дерево” |
| Tree.cpp | Файл, отвечающий за операции над деревом, такие как вставка, удаление и т.д. |
| Item.h | Заголовочный файла Item.cpp |
| Item.cpp | Файл, отвечающий за связь дерева с фигурами, которыми он будет заполняться |
| Iterator.h | Заголовочный файл, описывающий структуру итераторов |
| TStackItem.h | Заголовочный файла STackItem.cpp |
| TStackItem.cpp | Файл, описывающий функции элемента стека |
| TStack.h | Заголовочный файла STack.cpp |
| TStack.cpp | Файл, отвечающий за описание функций стека, в качестве контейнера 2-го уровня |
| TAllocatationBlock.h | Заголовочный файла TAllocatationBlock.cpp |
| TAllocatationBlock.cpp | Файл, отвечающий за реализацию аллокатора памяти |
| main.cpp | Модуль, отвечающий за “лицо” программы, содержащий интерфейс, позволяющий работать с написанным в других файлах кодом |

Пример выполнения программы

/home/chappybunny/CLionProjects/OOP8/cmake-build-debug/OOP8

TAllocationBlock: Memory init

Press 1 to get help

Press 2 to add figure in Tree

Press 3 to get a number of nodes on level

Press 4 to print Tree

Press 5 to delete figure from Tree

Press 6 to get demo iterator

Press 7 to get demo sort

Press 8 to get demo parallel sort

Press 0 to exit

2

Choose the figure:

1 - Pentagon

2 - Hexagon

3 - Octagon

3

Insert side of Octagon:

2

Insert the key of parent and the key of figure:

1 2

Tree was empty. Item was set as root.

2

Choose the figure:

1 - Pentagon

2 - Hexagon

3 - Octagon

3

Insert side of Octagon:

4

Insert the key of parent and the key of figure:

2 1

2

Choose the figure:

1 - Pentagon

2 - Hexagon

3 - Octagon

3

Insert side of Octagon:

1 5

Insert side of Octagon:

4

Insert the key of parent and the key of figure:

1 5

4

[Octagon with side = 2, Square = 10.3923, Key = 2]

[Octagon with side = 4, Square = 41.5692, Key = 1]

[Octagon with side = 4, Square = 41.5692, Key = 5]

8

4

[Octagon with side = 2, Square = 10.3923, Key = 1]

[Octagon with side = 4, Square = 41.5692, Key = 2]

[Octagon with side = 4, Square = 41.5692, Key = 5]

**main.cpp**

//Ваньков Денис М80-207Б-17

//Вариант 22(дерево, пятиугольник, шестиугольник, восьмиугольник)

#include <cstdlib>

#include <iostream>

#include <memory>

#include <cstdint>

#include "Pentagon.h"

#include "Hexagon.h"

#include "Octagon.h"

#include "Item.h"

#include "Tree.h"

#include "Figure.h"

void help(){

std::cout << "Press 1 to get help" << std::endl;

std::cout << "Press 2 to add figure in tree" << std::endl;

std::cout << "Press 3 to get a number of nodes on level" << std::endl;

std::cout << "Press 4 to print tree" << std::endl;

std::cout << "Press 5 to delete figure from tree" << std::endl;

std::cout << "Press 0 to exit" << std::endl;

}

int main(int argc, const char \* argv[]) {

size\_t act;

TTree<Figure> s;

help();

while (true) {

std::cin >> act;

if (act == 0) break;

if (act > 5) {

std::cout << "Error: enter another parameter (0-5)\n";

continue;

}

switch (act) {

case 1: {

help();

break;

}

case 2: {

size\_t type;

std::cout << "Choose the figure: \n";

std::cout << "1 - Pentagon\n";

std::cout << "2 - Hexagon\n";

std::cout << "3 - Octagon\n";

std::cin >> type;

if (type > 0) {

if (type > 3) {

std::cout << "Error: enter another parameter (0-3)\n";

continue;

}

switch (type) {

case 1: {

s.Push(std::shared\_ptr<TTreeItem<Figure>> (new TTreeItem<Figure> (std::shared\_ptr <Pentagon>(new Pentagon(std::cin)))));

break;

}

case 2: {

s.Push(std::shared\_ptr<TTreeItem<Figure>> (new TTreeItem<Figure> (std::shared\_ptr <Hexagon>(new Hexagon(std::cin)))));

break;

}

case 3: {

s.Push(std::shared\_ptr<TTreeItem<Figure>> (new TTreeItem<Figure> (std::shared\_ptr <Octagon>(new Octagon(std::cin)))));

break;

}

}

}

break;

}

case 3: {

int number = 0;

std::cout << "Enter level to get a number of nodes" << std::endl;

std::cin >> number;

std::cout << "On level " << number << " nodes: " << s.GetNodesOnLevel(number) << std::endl;

break;

}

case 4: {

std::cout << s << "\n";

break;

}

case 5: {

if (s.empty())

std::cout << "Tree is empty" << std::endl;

else {

std::cout << "Enter side of figure to delete it: " << std::endl;

size\_t value;

std::cin >> value;

s.Pop(value);

}

break;

}

case 6: {

if (s.empty())

std::cout << "Tree is empty" << std::endl;

else {

for (auto i : s) {

i->Print();

std::cout << std::endl;

}

}

break;

}

default: {

std::cerr << "Error: incorrect data\n";

break;

}

}

}

return 0;

}

**Tree.cpp**

#include "Tree.h"

template <class T>

Tree<T>::Tree()

{

this->root = nullptr;

}

template<class T>

TIterator<TreeItem<T>, T> Tree<T>::begin()

{

return TIterator<TreeItem<T>, T>(root);

}

template<class T>

TIterator<TreeItem<T>, T> Tree<T>::end()

{

return TIterator<TreeItem<T>, T>(nullptr);

}

template <class T>

std::shared\_ptr<TreeItem<T>> Tree<T>::insert(std::shared\_ptr<T> figure, size\_t parentKey, size\_t key)

{

std::shared\_ptr<TreeItem<T>> node = std::make\_shared<TreeItem<T>>(figure, key);

if (empty()) {

root = node;

root->setParent(nullptr);

std::cout << "Tree was empty. Item was set as root." << std::endl;

return root;

}

std::shared\_ptr<TreeItem<T>> parent = find(this->root, parentKey);

if (!parent) {

std::cout << "Parent with this key not found." << std::endl;

if (root->getSon()) {

std::shared\_ptr<TreeItem<T>> tmp = root->getSon();

while (tmp->getBrother())

tmp = tmp->getBrother();

tmp->setBrother(node);

node->setParent(root);

return tmp->getBrother();

} else {

root->setSon(node);

node->setParent(root);

return root->getSon();

}

}

if (parent->getSon()) {

std::shared\_ptr<TreeItem<T>> tmp = parent->getSon();

while (tmp->getBrother())

tmp = tmp->getBrother();

tmp->setBrother(node);

return tmp->getBrother();

} else {

parent->setSon(node);

node->setParent(parent);

return parent->getSon();

}

}

template <class T>

void Tree<T>::remove(size\_t key)

{

if (root->getKey() == key) {

root = nullptr;

}

else {

remove(root, key);

}

}

template <class T>

void Tree<T>::remove(std::shared\_ptr<TreeItem<T>> node, size\_t key)

{

if (node->getSon()) {

if (node->getSon()->getKey() == key) {

std::shared\_ptr<TreeItem<T>> tr = node->getSon();

node->setSon(node->getSon()->getBrother());

tr->setBrother(nullptr);

return;

} else {

remove(node->getSon(), key);

}

}

if (node->getBrother()) {

if (node->getBrother()->getKey() == key) {

std::shared\_ptr<TreeItem<T>> tr = node->getBrother();

node->setBrother(node->getBrother()->getBrother());

tr->setBrother(nullptr);

return;

} else {

remove(node->getBrother(), key);

}

}

}

template <class T>

bool Tree<T>::empty()

{

return this->root == nullptr;

}

template <class T>

std::shared\_ptr<TreeItem<T>> Tree<T>::find(std::shared\_ptr<TreeItem<T>> node, size\_t key)

{

std::shared\_ptr<TreeItem<T>> tr = nullptr;

if (node->getKey() == key)

return node;

if (node->getSon()) {

tr = find(node->getSon(), key);

if (tr != nullptr) {

return tr;

}

}

if (node->getBrother()) {

tr = find(node->getBrother(), key);

if (tr != nullptr) {

return tr;

}

}

return nullptr;

}

template <class T>

void Tree<T>::print()

{

this->print(this->root, 0);

}

template<class T>

size\_t Tree<T>::partition(size\_t array[], size\_t low, size\_t high)

{

size\_t middle = array[high]; // создаем массив размером равным максимальному значению элемента

size\_t i = (low - 1); // индекс меньшего элемента

for (size\_t j = low; high != 0 && j <= high - 1; j++)

{

if (array[j] <= middle)

{

i++; // Увеличиваем index минимального

size\_t tmp = array[i];

array[i] = array[j];

array[j] = tmp;

}

}

size\_t tmp = array[i + 1];

array[i + 1] = array[high];

array[high] = tmp;

std::cout << i + 1 << std::endl;

return (i + 1);

}

template <class T>

void Tree<T>::print(std::shared\_ptr<TreeItem<T>> tree, size\_t spacing)

{

if (tree)

for (size\_t i = 0; i < spacing; i++)

std::cout << "\t";

tree->print();

if (tree->getSon())

print(tree->getSon(), spacing + 1);

if (tree->getBrother())

print(tree->getBrother(), spacing);

}

template<class T>

void Tree<T>::quickSort(size\_t arr[], size\_t low, size\_t high)

{

if (low < high)

{

size\_t pi = partition(arr, low, high);

if (pi > 0)

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

template<class T>

void Tree<T>::sort()

{

size\_t size = 0;

for (auto it : \*this) {

size++;

}

size\_t arr[100];

auto it = begin();

for (size\_t i = 0; i < size; ++i) {

arr[i] = it->getKey();

it++;

}

it = begin();

quickSort(arr, 0, size - 1);

for (size\_t i = 0; i < size; ++i) {

it->setKey(arr[i]);

it++;

}

}

size\_t futureQuickSort(size\_t \*arr, size\_t left, size\_t right)

{

size\_t i = left, j = right;

size\_t tmp;

size\_t pivot = arr[(left + right) / 2];

while (i <= j) {

while (arr[i] < pivot)

i++;

while (arr[j] > pivot)

j--;

if (i <= j) {

tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

i++;

j--;

}

}

if (i < right) {

std::packaged\_task<size\_t(size\_t \*, size\_t, size\_t)> task(futureQuickSort);

auto result = task.get\_future();

std::thread task\_td(std::move(task), std::ref(arr), i, right);

task\_td.join();

result.get();

}

if (left < j) {

std::packaged\_task<size\_t(size\_t \*, size\_t, size\_t)> task(futureQuickSort);

auto result = task.get\_future();

std::thread task\_td(std::move(task), std::ref(arr), left, j);

task\_td.join();

result.get();

}

return 0;

}

template<class T>

void Tree<T>::sortParallel()

{

size\_t size = 0;

for (auto it : \*this) {

size++;

}

size\_t arr[100];

auto it = begin();

for (size\_t i = 0; i < size; ++i) {

arr[i] = it->getKey();

it++;

}

futureQuickSort(arr, 0, size - 1);

it = begin();

for (size\_t i = 0; i < size; ++i) {

it->setKey(arr[i]);

it++;

}

}

template <class T>

Tree<T>::~Tree()

{}

template <class T>

std::ostream & operator<<(std::ostream & os, Tree<T> & obj)

{

obj.print();

return os;

}

#include "Figure.h"

template class Tree<Figure>;

//template std::ostream& operator<<(std::ostream& os, const Tree<Figure>& Tree);

**Tree.h**

#ifndef TREE\_H

#define TREE\_H

#include "TreeItem.h"

#include "iterator.h"

#include <memory>

#include <mutex>

#include <future>

template <class T> class Tree

{

public:

Tree();

TIterator<TreeItem<T>, T> begin();

TIterator<TreeItem<T>, T> end();

std::shared\_ptr<TreeItem<T>> insert(std::shared\_ptr<T> figure, size\_t parentKey, size\_t key);

void remove(size\_t key);

void remove(std::shared\_ptr<TreeItem<T>> tree, size\_t key);

bool empty();

std::shared\_ptr<TreeItem<T>> find(std::shared\_ptr<TreeItem<T>> tree, size\_t key);

void print();

void print(std::shared\_ptr<TreeItem<T>> tree, size\_t spacing);

//template <class A> friend std::ostream& operator<<(std::ostream& os, const Tree<A> & obj);

void sort();

void sortParallel();

~Tree();

private:

// std::future<void> sortInBackground();

size\_t partition(size\_t array[], size\_t low, size\_t high);

void quickSort(size\_t arr[], size\_t low, size\_t high);

std::shared\_ptr<TreeItem<T>> root;

};

#endif // !TREE\_H

**Item.h**

#ifndef OOP\_LAB4\_ITEM\_H

#define OOP\_LAB4\_ITEM\_H

#include <stdio.h>

#include <memory>

#include "Figure.h"

template <class T>

class TTreeItem {

public:

TTreeItem(const std::shared\_ptr<T> &figure);

template <class A>

friend std::ostream& operator<<(std::ostream &os, const TTreeItem<A> &obj);

std::shared\_ptr<TTreeItem<T>> GetSon();

std::shared\_ptr<TTreeItem<T>> GetBrother();

void SetSon(std::shared\_ptr<TTreeItem<T>> son);

void SetBrother(std::shared\_ptr<TTreeItem<T>> brother);

std::shared\_ptr<T> GetFigure() const;

private:

std::shared\_ptr<T> figure;

std::shared\_ptr<TTreeItem<T>> son;

std::shared\_ptr<TTreeItem<T>> brother;

};

#endif //OOP\_LAB4\_ITEM\_H

**Item.cpp**

#include <iostream>

#include <cstdlib>

#include "Item.h"

#include "Figure.h"

template <class T>

TTreeItem<T>::TTreeItem(const std::shared\_ptr<T> &figure) {

this->figure = figure;

this->son = nullptr;

this->brother = nullptr;

std::cout << "Tree item: created" << std::endl;

}

template <class T>

std::shared\_ptr<T> TTreeItem<T>::GetFigure() const {

return this->figure;

}

template <class T>

std::shared\_ptr<TTreeItem<T>> TTreeItem<T>::GetSon() {

return this->son;

}

template <class T>

void TTreeItem<T>::SetSon(std::shared\_ptr<TTreeItem<T>> son) {

this->son = son;

}

template <class T>

void TTreeItem<T>::SetBrother(std::shared\_ptr<TTreeItem<T>> brother) {

this->brother = brother;

}

std::shared\_ptr<TTreeItem<T>> TTreeItem<T>::GetNext()

{

std::shared\_ptr<TTreeItem<T>> tmp;

if (this->GetSon()) {

return GetSon();

}

else if (this->GetBrother()) {

return GetBrother();

} else if (this->GetParent()) {

tmp = this->GetParent();

while (tmp && !(tmp->GetBrother())) {

tmp = tmp->GetParent();

}

if (!tmp)

return nullptr;

tmp = tmp->GetBrother();

return tmp;

}

return nullptr;

}

template <class T>

std::shared\_ptr<TTreeItem<T>> TTreeItem<T>::GetBrother() {

return this->brother;

}

template <class A>

std::ostream& operator<<(std::ostream &os, const TTreeItem<A> &obj) {

os << "[" << obj.GetFigure() << " " << obj.GetFigure()->Square() << "]";

return os;

}

**Iterator.h**

#ifndef OOP\_LAB4\_ITERATOR\_H

#define OOP\_LAB4\_ITERATOR\_H

#include <memory>

#include <iostream>

template <class N, class T>

class TIterator

{

public:

TIterator(std::shared\_ptr<N> n) {

cur = n;

}

std::shared\_ptr<T> operator\* () {

return cur->GetFigure();

}

std::shared\_ptr<T> operator-> () {

return cur->GetFigure();

}

void operator++() {

cur = cur->GetNext();

}

TIterator const operator++ (int) {

TIterator cur(\*this);

++(\*this);

return cur;

}

bool operator== (const TIterator &it) {

return (cur == it.cur);

}

bool operator!= (const TIterator &it) {

return (cur != it.cur);

}

private:

std::shared\_ptr<N> cur;

};

#endif //OOP\_LAB4\_ITERATOR\_H

**TAllocationBlock.h**

#ifndef OOP\_LAB4\_TALLOCATIONBLOCK\_H

#define OOP\_LAB4\_TALLOCATIONBLOCK\_H

#include <iostream>

#include <cstdlib>

#include "TStack.h"

typedef unsigned char Byte;

class TAllocationBlock

{

public:

TAllocationBlock(size\_t size, size\_t count);

void \*Allocate();

void Deallocate(void \*ptr);

bool Empty();

size\_t Size();

virtual ~TAllocationBlock();

private:

Byte \*\_used\_blocks;

TStack<void \*>\_free\_blocks;

};

#endif //OOP\_LAB4\_TALLOCATIONBLOCK\_H

**TAllocationBlock.cpp**

#include "TAllocationBlock.h"

TAllocationBlock::TAllocationBlock(size\_t size, size\_t count) {

\_used\_blocks = (Byte \*)malloc(size \* count);

for (size\_t i = 0; i < count; ++i) {

void \*ptr =(void \*)malloc(sizeof(void \*));

ptr = \_used\_blocks + i \* size;

\_free\_blocks.Push(ptr);

}

}

void \*TAllocationBlock::Allocate() {

if(!\_free\_blocks.IsEmpty()) {

void \*res = \_free\_blocks.Top();

\_free\_blocks.Pop();

return res;

} else {

throw std::bad\_alloc();

}

}

void TAllocationBlock::Deallocate(void \*ptr) {

\_free\_blocks.Push(ptr);

}

bool TAllocationBlock::Empty() {

return \_free\_blocks.IsEmpty();

}

size\_t TAllocationBlock::Size() {

return \_free\_blocks.GetSize();

}

TAllocationBlock::~TAllocationBlock() {

while(!\_free\_blocks.IsEmpty()) {

\_free\_blocks.Pop();

}

free(\_used\_blocks);

//delete \_free\_blocks;

//delete \_used\_blocks;

}

#ifndef OOP\_LAB4\_TSTACK\_H

#define OOP\_LAB4\_TSTACK\_H

**TStack.h**

#include <iostream>

#include <memory>

#include "TStackItem.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:

TStackItem<T> \*head;

uint32\_t count;

};

#endif //OOP\_LAB4\_TSTACK\_H

**TStack.cpp**

#include <iostream>

#include <memory>

#include "TStackItem.h"

template <class T>

TStackItem<T>::TStackItem(const T &val, TStackItem<T> \*item) {

value = new T(val);

next = item;

}

template <class T>

void TStackItem<T>::Push(const T &val) {

\*value = val;

}

template <class T>

T &TStackItem<T>::Pop() const {

return \*value;

}

template <class T>

void TStackItem<T>::SetNext(TStackItem<T> \*item) {

next = item;

}

template <class T>

TStackItem<T> &TStackItem<T>::GetNext() const {

return \*next;

}

template <class T>

TStackItem<T>::~TStackItem() {

delete value;

}

typedef unsigned char Byte;

template class

TStackItem<void \*>;

Вывод

Распараллеливание программы - это нелегкая задача для разработчика, намного по сложности превосходящая программирование классическое. Для начала программист должен решить, какие части программы могут выполняться одновременно, что выбрать схему параллелизации, и как эти части будут взаимодействовать.

Одна из самых сложных проблем при разработке параллельных программ - это отладка. Дело в том, что два раза запустив программу в таком режиме, мы запросто можем получить разные результаты. Так как части кода, выполняющиеся независимо, могут выполняться с разной скоростью, в зависимости от того, кто получил больше времени от операционной системы. И из-за этого разные события могут наступать в разное время, приводя к тому, что проблема иногда возникает, а иногда, что интересно - нет.

Широкое распространение параллельные вычисления приобрели с переходом компьютерной индустрии на массовый выпуск многоядерных процессоров с векторными расширениями. В настоящие время практически все устройства, от смартфонов и до самых мощных суперкомпьютеров – оборудованы многоядерными процессорами. Также если вы пишете последовательную программу, не применив распределение работы между разными ядрами центрального процессора и не проведя векторизацию, то вы используете только часть вычислительных возможностей центрального процессора, и не задействуете его на полную мощьность.

Список литературы

1. Бьерн Страуструп — “Язык программирования С++” (Издание 3 специальное).
2. Справочник по языку C++ [Электронный ресурс]. URL: http://cppstudio.com .
3. Википедия (Электронный ресурс).

URL: https://ru.wikipedia.org/wiki/Инкапсуляция\_(программирование)