МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Белорусский Государственный Технологический Университет

**РЕФЕРАТ**

по дисциплине: «Основы Алгоритмизации и Программирования»

на тему: “ Отличие объектов, созданных разными структурами данных”

Выполнил студент группы

Оглавление

[Массив(Array) 2](#_Toc158319708)

[Вектор(Vector) 3](#_Toc158319709)

[Список(List) 4](#_Toc158319710)

[Стек(Stack) 4](#_Toc158319711)

[Очередь(Queue) 5](#_Toc158319712)

[Дерево(Tree) 6](#_Toc158319713)

[Граф(Graph) 8](#_Toc158319714)

[Хеш-таблица(Hash Table) 8](#_Toc158319715)

[Сравнение объектов, созданных различными структурами данных 9](#_Toc158319716)

# Массив(Array)

**Массив** - это структура данных, представленная в виде группы ячеек одного типа, объединенных под одним единым именем.

**Применение**: Массивы используются для обработки большого количества однотипных данных.

Пример:

#include <iostream>

int main() {

int numbers[5] = { 1, 2, 3, 4, 5 };

// Доступ к элементам массива по индексу

std::cout << "Third element: " << numbers[2] << std::endl;

// Изменение значения элемента

numbers[1] = 10;

std::cout << "Array: ";

for (int i = 0; i < 5; ++i) {

std::cout << numbers[i] << " ";

}

return 0;

}



# Вектор(Vector)

Вектор в C++ - это динамический массив, предоставляемый стандартной библиотекой (STL). **В отличие от обычного массива, вектор имеет динамический размер и обеспечивает удобное управление памятью**.

**Применение**: Векторы используются для хранения и обработки динамических последовательностей данных, где необходимо эффективное добавление и удаление элементов.

Пример:

#include <iostream>

#include <vector>

int main() {

// Создание вектора чисел

std::vector<int> numbers = { 1, 2, 3, 4, 5 };

// Доступ к элементам вектора по индексу

std::cout << "Third element: " << numbers[2] << std::endl;

// Изменение значения элемента

numbers[1] = 10;

// Добавление элемента в конец вектора

numbers.push\_back(6);

// Вывод всех элементов вектора

std::cout << "Vector: ";

for (const auto& num : numbers) {

std::cout << num << " ";

}

return 0;

}



**Примечание**:

Векторы в C++ обладают множеством удобных методов, таких как push\_back, pop\_back, size, и другие. Они предоставляют динамическое управление размером и автоматическое управление памятью, что делает их удобными для работы с изменяющимися данными.

# Список(List)

**Список в C++ представляет собой структуру данных, представленную в виде двусвязного списка узлов**. Каждый узел содержит данные и указатели на предыдущий и следующий узлы.

**Применение**: Списки используются для эффективного добавления и удаления элементов в середине структуры данных, также они обеспечивают гибкость в управлении памятью.

Пример:

#include <iostream>

#include <list>

int main() {

// Создание списка чисел

std::list<int> numbers = { 1, 2, 3, 4, 5 };

// Вставка элемента после второго элемента списка

auto it = std::next(numbers.begin(), 1);

numbers.insert(it, 6);

// Удаление третьего элемента списка

it = std::next(numbers.begin(), 2);

numbers.erase(it);

std::cout << "List: ";

for (const auto& num : numbers) {

std::cout << num << " ";

}

return 0;

}



**Примечание**:

Списки эффективны тем, что позволяют вставлять и удалять элементы из середины списка, но могут быть менее эффективны, чем массивы или векторы, поскольку доступ к элементам осуществляется последовательно. Структура двусвязных списков позволяет эффективно выполнять операции вставки и удаления, что делает списки подходящими для различных сценариев.

# Стек(Stack)

**Стек в C++ - это структура данных, которая работает по принципу LIFO (last in, first out).** Это означает, что последний элемент, добавленный в стек, вынимается первым.

**Применение**: Стеки полезны в ситуациях, когда необходимо отслеживать порядок обработки данных, например при реализации рекурсивных алгоритмов или управлении вызовами функций.

Пример:

#include <iostream>

#include <stack>

int main() {

// Создание стека чисел

std::stack<int> numbers;

// Добавление элементов в стек

numbers.push(1);

numbers.push(2);

numbers.push(3);

// Вывод верхнего элемента стека

std::cout << "Top element: " << numbers.top() << std::endl;

// Удаление верхнего элемента

numbers.pop();

// Проверка, пуст ли стек

std::cout << "Is stack empty? " << (numbers.empty() ? "Yes" : "No") << std::endl;

return 0;

}



**Примечание**:

Стеки подходят для определенных ситуаций, таких как вызовы функций и алгоритмическое управление состоянием, поскольку они обеспечивают эффективные операции добавления и удаления элементов с одного конца.

# Очередь(Queue)

**Очередь в C++ представляет собой структуру данных, работающую по принципу "First In , First Out" (FIFO).** Это означает, что первый элемент, добавленный в очередь, будет первым, который будет извлечен.

**Применение**: Очереди часто используются там, где необходимо управлять элементами в порядке, в котором они приходят, например, в системах управления заданиями, обработке запросов и т.д.

Пример:

#include <iostream>

#include <queue>

int main() {

// Создание очереди чисел

std::queue<int> numbers;

// Добавление элементов в очередь

numbers.push(1);

numbers.push(2);

numbers.push(3);

// Вывод переднего элемента очереди

std::cout << "Front element: " << numbers.front() << std::endl;

// Удаление переднего элемента

numbers.pop();

// Проверка, пуста ли очередь

std::cout << "Is queue empty? " << (numbers.empty() ? "Yes" : "No") << std::endl;

return 0;

}



**Примечание**:

Очередь является удобной структурой данных для сценариев, где порядок поступления элементов важен. Она часто используется для управления задачами в операционных системах, а также в сетевых и параллельных вычислениях.

# Дерево(Tree)

**Дерево в программировании представляет собой иерархическую структуру данных, состоящую из узлов. Каждый узел может иметь одного родителя и ноль или более дочерних узлов**.

**Применение**: Деревья используются для организации данных в иерархической форме, представления отношений иерархии, поиска и сортировки данных.

**Узлы дерева**: Каждый узел дерева состоит из данных (ключа) и ссылок на дочерние узлы (поддеревья). В деревьях бинарного поиска (BST), каждый узел имеет не более двух дочерних узлов: левого и правого.

Пример:

#include <iostream>

// Определение структуры для узла бинарного дерева

struct TreeNode {

int data;

TreeNode\* left;

TreeNode\* right;

// Конструктор для удобного создания узлов

TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}

};

TreeNode\* insert(TreeNode\* root, int value) {

// Если узел пуст, создаем новый узел с переданным значением

if (root == nullptr) {

return new TreeNode(value);

}

// Рекурсивно ищем место для вставки в левое или правое поддерево

if (value < root->data) {

root->left = insert(root->left, value);

}

else if (value > root->data) {

root->right = insert(root->right, value);

}

return root; // Возвращаем указатель на измененный узел

}

// Рекурсивная функция для обхода дерева

void inOrderTraversal(TreeNode\* node) {

if (node != nullptr) {

// Рекурсивно обходим левое поддерево

inOrderTraversal(node->left);

// Выводим значение текущего узла

std::cout << node->data << " ";

// Рекурсивно обходим правое поддерево

inOrderTraversal(node->right);

}

}

int main() {

// Создание бинарного дерева поиска

TreeNode\* root = nullptr;

root = insert(root, 10);

insert(root, 5);

insert(root, 15);

insert(root, 3);

insert(root, 7);

// Проход по дереву

std::cout << "In-order traversal: ";

inOrderTraversal(root);

std::cout << std::endl;

return 0;

}



**Примечание:** Бинарные деревья могут использоваться для эффективного поиска, вставки и удаления данных. Различные типы деревьев, такие как AVL-деревья или красно-черные деревья, могут обеспечивать балансировку для оптимизации производительности.

# Граф(Graph)

**Граф в программировании представляет собой коллекцию узлов (вершин) и рёбер (ребра), которые соединяют эти узлы**. Узлы могут представлять сущности, а рёбра - отношения между этими сущностями.

**Применение**: Графы используются для моделирования отношений между объектами в различных областях, таких как социальные сети, транспортные системы, вычислительные сети и многие другие.

**Вершины и рёбра графа**: Вершины представляют объекты, а рёбра - отношения между этими объектами. Граф может быть направленным (рёбра имеют направление) или ненаправленным.

# Хеш-таблица(Hash Table)

В программировании **хэш-таблица - это структура данных, которая использует хэш-функции для сопоставления данных с индексами (слотами) в массиве для быстрого поиска**.

**Области применения**: Хеш-таблицы широко используются при реализации ассоциативных массивов, множеств и других структур данных, требующих эффективного поиска, вставки и удаления элементов.

**Хэш-функции**: Хэш-функции принимают входные данные и генерируют хэш-код фиксированного размера. Хорошие хэш-функции стремятся равномерно распределить данные по слотам, чтобы минимизировать коллизии (случай, когда два разных фрагмента данных соответствуют одному и тому же слоту).

# Сравнение объектов, созданных различными структурами данных

**Таким образом объект, созданный разными структурами данных имеет разное применение, разный функционал, разный синтаксис.**

**Массивы:**

Применение: Хранение последовательности элементов одного типа с прямым доступом по индексу.

Функционал: Эффективный доступ и изменение элементов, поддержка множества операций (сортировка, поиск и т.д.).

**Векторы**:

Применение: Динамический массив, предоставляющий быстрые операции вставки и удаления в конец, а также произвольный доступ по индексу.

Функционал: Динамическое изменение размера, эффективный доступ, поддержка множества операций, таких как вставка, удаление, сортировка.

**Списки**:

Применение: Упорядоченные коллекции элементов разных типов, динамическое изменение размера.

Функционал: Вставка/удаление в середине списка, простое добавление/удаление элементов.

**Стеки**:

Применение: Хранение элементов с доступом только к вершине, работа по принципу "Last In, First Out" (LIFO).

Функционал: Вставка и удаление элементов с вершины, поддержка рекурсии.

**Очереди**:

Применение: Хранение элементов с доступом к первому вставленному элементу, работа по принципу "First In, First Out" (FIFO)

Функционал: Вставка в конец и удаление из начала, использование в задачах с ожиданием (например, управление задачами).

**Деревья**:

Применение: Организация данных в иерархической структуре, поиск, сортировка, представление иерархической информации.

Функционал: Эффективный поиск, вставка, удаление, различные виды обхода (ин-ордер, пре-ордер и т.д.).

**Графы**:

Применение: Представление отношений между объектами, моделирование сетей, социальных связей и т.д.

Функционал: Обходы (DFS, BFS), поиск путей, алгоритмы на основе графов (кратчайший путь, минимальное остовное дерево и т.д.).

**Хеш-таблицы**:

Применение: Реализация ассоциативных массивов, быстрый поиск, вставка и удаление.

Функционал: Эффективное распределение данных по хеш-функции, обработка коллизий, поиск в среднем за константное время.

# Литература:

1. <https://brestprog.by/topics/graphs/>
2. <https://habr.com/ru/articles/267855/>
3. <https://habr.com/ru/articles/509220/>
4. <http://cppstudio.com/post/389/>
5. <https://metanit.com/cpp/tutorial/7.11.php>
6. <https://habr.com/ru/articles/341586/>
7. <https://metanit.com/cpp/tutorial/7.6.php>