**Белорусский государственный технологический университет**

**Факультет информационных технологий**

**Кафедра программной инженерии**

**Реферат “** **Оптимизаторы, маршрутизаторы сложных структур данных и контейнеры STL библиотек для языка С++ 17** **”**

По дисциплине «Основы алгоритмизации и программирования»

Выполнил:

Студент 1 курса 10 группы ПИ

Макаревич Кирилл Витальевич

Проверил: Белодед Николай Иванович

Оглавление

[1. Оптимизаторы в C++ 3](#_Toc194507555)

[1.1. Введение в оптимизацию программного кода 3](#_Toc194507556)

[1.2. Компиляторные оптимизации 3](#_Toc194507557)

[1.2.1. Основные техники компиляторной оптимизации 3](#_Toc194507558)

[1.2.2. Преимущества и ограничения компиляторных оптимизаций 5](#_Toc194507559)

[1.3. Ручные оптимизации 5](#_Toc194507560)

[1.3.1. Выбор эффективных алгоритмов 5](#_Toc194507561)

[1.3.2. Минимизация копирования данных 6](#_Toc194507562)

[1.3.3. Кэширование результатов 6](#_Toc194507563)

[1.3.4. Оптимизация работы с памятью 6](#_Toc194507564)

[1.4. Взаимодействие ручных и компиляторных оптимизаций 6](#_Toc194507565)

[2. Маршрутизаторы сложных структур данных 6](#_Toc194507566)

[2.1. Определение и назначение 6](#_Toc194507567)

[2.2. Примеры структур данных 7](#_Toc194507568)

[2.3. Алгоритмы маршрутизации 7](#_Toc194507569)

[2.3.1. Алгоритм Дейкстры для графов 7](#_Toc194507570)

[3. Контейнеры STL в C++17 8](#_Toc194507571)

[3.1. Роль STL в программировании 8](#_Toc194507572)

[3.2. Категории контейнеров 8](#_Toc194507573)

[3.3. Нововведения C++17 9](#_Toc194507574)

[3.3.1. std::string\_view 9](#_Toc194507575)

[3.3.2. std::optional 9](#_Toc194507576)

[Заключение 10](#_Toc194507577)

## 

## 

## 1. Оптимизаторы в C++

### 1.1. Введение в оптимизацию программного кода

Оптимизация в программировании — это процесс улучшения характеристик программы, таких как скорость выполнения, использование памяти или потребление энергии, без изменения её функциональности. В контексте C++ оптимизация особенно важна, так как этот язык часто используется для создания высокопроизводительных систем, таких как игровые движки, операционные системы и научные симуляции. Оптимизаторы в C++ можно разделить на два основных типа: автоматические (выполняемые компилятором) и ручные (реализуемые разработчиком). Оба подхода имеют свои преимущества и ограничения, и их совместное использование позволяет достичь максимальной эффективности.

C++ как язык предоставляет разработчикам низкоуровневый контроль над ресурсами, что делает его идеальным для оптимизации, но одновременно требует глубокого понимания как языка, так и аппаратного обеспечения. Современные компиляторы, такие как GCC (GNU Compiler Collection), Clang и MSVC (Microsoft Visual C++), включают мощные механизмы оптимизации, которые могут значительно улучшить производительность кода без вмешательства программиста. Однако в некоторых случаях ручная оптимизация остаётся необходимой, особенно когда автоматические методы не могут учесть специфику задачи.

### 1.2. Компиляторные оптимизации

Компиляторные оптимизации — это преобразования исходного кода, выполняемые на этапе компиляции, чтобы сделать результирующий машинный код более эффективным. Компиляторы C++ поддерживают несколько уровней оптимизации, которые активируются с помощью флагов, таких как -O1, -O2, -O3 в GCC и Clang или /O2 в MSVC. Каждый уровень включает всё более сложные и агрессивные техники, хотя это может увеличивать время компиляции и усложнять отладку.

### 1.2.1. Основные техники компиляторной оптимизации

Рассмотрим подробнее ключевые методы, которые применяются компиляторами для улучшения кода.

1. **Инлайн-подстановка функций (Function Inlining)**  
   Инлайн-подстановка — это процесс замены вызова функции её непосредственным телом в месте вызова. Это устраняет накладные расходы, связанные с передачей управления (например, сохранение состояния стека и возврат результата), что особенно полезно для небольших функций, вызываемых часто. Однако чрезмерное использование инлайнинга может увеличить размер исполняемого файла, что иногда негативно сказывается на производительности из-за кэш-памяти процессора.

* **Пример кода на C++:**

|  |
| --- |
| * inline int multiply(int a, int b) {  return a \* b; }  int main() {  int result = multiply(5, 3); // Компилятор заменяет вызов на 5 \* 3  return result; } |

* **Псевдокод:**
* функция multiply(a, b):  
   вернуть a \* b  
    
  главная\_функция:  
   result = multiply(5, 3) // заменяется на result = 15  
   вернуть result
* В этом примере компилятор с флагом -O2 или выше, скорее всего, заменит вызов multiply(5, 3) на прямое вычисление 5 \* 3, что сократит время выполнения программы. Ключевое слово inline подсказывает компилятору, что функция является кандидатом для подстановки, хотя окончательное решение принимает сам компилятор.

1. **Удаление мёртвого кода (Dead Code Elimination)**  
   Мёртвый код — это участки программы, которые не влияют на её результат. Компилятор анализирует код и удаляет такие фрагменты, что уменьшает размер программы и ускоряет её выполнение. Этот метод особенно полезен при использовании условных операторов с константными условиями.

* **Пример кода на C++:**

|  |
| --- |
| * int main() {  int x = 10;  **if** (**false**) {  x = 20; *// Этот код удаляется компилятором*  }  **return** x; } |

* **Псевдокод:**
* главная\_функция:  
   x = 10  
   если ложь:  
   x = 20 // удаляется  
   вернуть x
* Здесь условие if (false) никогда не выполняется, поэтому компилятор полностью исключает ветку с присваиванием x = 20. Это простейший пример, но в реальных проектах удаление мёртвого кода может затрагивать большие участки программы, например, неиспользуемые переменные или функции.

1. **Константное свёртывание (Constant Folding)**  
   Константное свёртывание — это техника, при которой компилятор вычисляет значения константных выражений на этапе компиляции, а не во время выполнения программы. Это снижает количество операций, выполняемых процессором.

* **Пример кода на C++:**

|  |
| --- |
| * int main() {  int a = 5 + 7; // Компилятор заменяет на int a = 12  return a; } |

* **Псевдокод:**
* главная\_функция:  
   a = 5 + 7 // заменяется на a = 12  
   вернуть a
* В данном случае компилятор видит, что 5 + 7 — это константное выражение, и заменяет его на результат 12. Это особенно полезно в сложных математических вычислениях, где много промежуточных константных значений.

1. **Оптимизация циклов (Loop Optimization)**  
   Циклы — одна из самых ресурсоёмких частей программ, поэтому компиляторы применяют к ним специальные оптимизации, такие как размотка циклов (loop unrolling) и вынесение инвариантных выражений (loop-invariant code motion). Размотка циклов уменьшает количество проверок условия выхода из цикла, а вынесение инвариантов убирает повторные вычисления внутри цикла.

* **Пример кода на C++:**

|  |
| --- |
| int sum = 0;  for (int i = 0; i < 4; ++i) {  sum += i;  } |

* Компилятор может преобразовать этот код в:
* int sum = 0;  
  sum += 0;  
  sum += 1;  
  sum += 2;  
  sum += 3;
* **Псевдокод:**
* sum = 0  
  для i от 0 до 3:  
   sum = sum + i  
  // Разворачивается в:  
  sum = 0  
  sum = sum + 0  
  sum = sum + 1  
  sum = sum + 2  
  sum = sum + 3
* Размотка цикла устраняет накладные расходы на управление счётчиком i и проверку условия i < 4. Однако для больших циклов полная размотка невозможна, и компилятор может развернуть только часть итераций.

### 1.2.2. Преимущества и ограничения компиляторных оптимизаций

Компиляторные оптимизации имеют ряд преимуществ:

* **Автоматизация**: разработчику не нужно вручную переписывать код.
* **Точность**: компилятор учитывает архитектуру процессора и применяет оптимизации, недоступные человеку.
* **Скорость**: многие преобразования выполняются быстрее, чем ручная работа.

Однако есть и ограничения:

* **Непредсказуемость**: компилятор может не применить оптимизацию, если посчитает её неэффективной.
* **Усложнение отладки**: оптимизированный код может не соответствовать исходному, что затрудняет поиск ошибок.
* **Зависимость от флагов**: без указания уровня оптимизации (например, -O2) код остаётся неоптимизированным.

### 1.3. Ручные оптимизации

Ручные оптимизации требуют от разработчика активного участия и глубокого анализа программы. Они применяются там, где компилятор не может справиться самостоятельно или когда нужно учесть специфику задачи.

### 1.3.1. Выбор эффективных алгоритмов

Одним из самых мощных способов ручной оптимизации является выбор подходящего алгоритма. Например, замена пузырьковой сортировки (O(n²)) на быструю сортировку (O(n log n)) может ускорить программу в сотни раз при больших объёмах данных. В C++ это часто реализуется с использованием стандартных функций, таких как std::sort, которые уже оптимизированы.

### 1.3.2. Минимизация копирования данных

Копирование больших объектов в C++ может быть дорогостоящим по времени и памяти. Использование ссылок и указателей позволяет избежать ненужных копий.

**Пример кода на C++:**

|  |
| --- |
| #include **<vector>**  void process(const std::vector<int>& data) { *// Константная ссылка*  **for** (const **auto**& value : data) {  *// Обработка без копирования*  } }  int main() {  std::vector<int> vec = {1, 2, 3};  process(vec);  **return** 0; } |

**Псевдокод:**

функция process(data):  
 для каждого value в data:  
 обработать value  
  
главная\_функция:  
 vec = [1, 2, 3]  
 process(vec)

Здесь const std::vector<int>& передаёт вектор по ссылке, избегая его копирования. Это особенно важно для больших контейнеров, где копирование может занять значительное время.

### 1.3.3. Кэширование результатов

Если программа часто выполняет одни и те же вычисления, их результаты можно сохранить в памяти (кэшировать), чтобы избежать повторных операций. Например, в вычислении чисел Фибоначчи кэширование позволяет снизить сложность с экспоненциальной до линейной.

### 1.3.4. Оптимизация работы с памятью

C++ позволяет управлять памятью вручную, что даёт возможность минимизировать фрагментацию и улучшить использование кэша процессора. Например, использование std::vector вместо std::list для последовательного доступа к данным ускоряет выполнение за счёт локальности памяти.

### 1.4. Взаимодействие ручных и компиляторных оптимизаций

Ручные и компиляторные оптимизации не исключают друг друга, а дополняют. Например, разработчик может написать код с использованием ссылок для минимизации копирования, а компилятор дополнительно применит инлайн-подстановку к функциям внутри этого кода. Однако важно избегать “избыточной оптимизации”, когда попытки улучшить код вручную мешают компилятору применять свои методы.

## 2. Маршрутизаторы сложных структур данных

### 2.1. Определение и назначение

Маршрутизаторы сложных структур данных — это алгоритмы и механизмы, которые управляют доступом, поиском и обработкой данных в структурах, таких как графы, деревья и хеш-таблицы. Эти структуры используются в задачах, требующих сложной организации данных, например, в сетевых протоколах, базах данных и системах искусственного интеллекта. В C++ маршрутизация часто реализуется с использованием контейнеров STL, что упрощает разработку и повышает производительность.

### 2.2. Примеры структур данных

1. **Графы**  
   Графы представляют собой набор вершин, соединённых рёбрами. Они могут быть направленными или ненаправленными, взвешенными или невзвешенными. В C++ графы часто реализуются через списки смежности с использованием std::vector<std::list<int>> или матрицы смежности с std::vector<std::vector<bool>>.
2. **Деревья**  
   Деревья — это иерархические структуры, где каждая вершина (узел) имеет не более одного родителя. Бинарные деревья поиска, например, используются в контейнерах std::set и std::map, которые реализуют красно-чёрные деревья для обеспечения логарифмической сложности операций.
3. **Хеш-таблицы**  
   Хеш-таблицы обеспечивают быстрый доступ к данным по ключу с амортизированной сложностью O(1). В C++ они представлены контейнерами std::unordered\_map и std::unordered\_set.

### 2.3. Алгоритмы маршрутизации

### 2.3.1. Алгоритм Дейкстры для графов

Алгоритм Дейкстры находит кратчайший путь от одной вершины до всех остальных в взвешенном графе с неотрицательными весами. Он широко применяется в сетевых маршрутизаторах и системах навигации.

**Псевдокод:**

функция dijkstra(граф, старт):  
 создать distances с бесконечностью для всех вершин  
 distances[старт] = 0  
 создать очередь\_с\_приоритетом pq  
 добавить (0, старт) в pq  
  
 пока pq не пуста:  
 u = извлечь вершину с минимальным расстоянием из pq  
 для каждого соседа v из u:  
 alt = distances[u] + вес(u, v)  
 если alt < distances[v]:  
 distances[v] = alt  
 добавить (alt, v) в pq  
 вернуть distances

**Пример кода на C++:**

|  |
| --- |
| using Graph = std::vector<std::vector<std::pair<int, int>>>;  std::vector<int> dijkstra(const Graph& graph, int start) {  int n = graph.size();  std::vector<int> distances(n, std::numeric\_limits<int>::max());  distances[start] = 0;  std::priority\_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, std::greater<>> pq;  pq.push({ 0, start });  while (!pq.empty()) {  int u = pq.top().second;  int dist\_u = pq.top().first;  pq.pop();  if (dist\_u > distances[u]) continue;  for (const auto& edge : graph[u]) {  int v = edge.first;  int weight = edge.second;  int alt = distances[u] + weight;  if (alt < distances[v]) {  distances[v] = alt;  pq.push({ alt, v });  }  }  }  return distances;  }  int main() {  int n = 4;  Graph graph(n);  graph[0].push\_back({ 1, 4 });  graph[0].push\_back({ 2, 1 });  graph[1].push\_back({ 3, 1 });  graph[2].push\_back({ 1, 2 });  graph[2].push\_back({ 3, 5 });  graph[3].push\_back({ 0, 1 });  std::vector<int> distances = dijkstra(graph, 0);  for (int i = 0; i < n; ++i) {  std::cout << "Distance to " << i << ": " << distances[i] << std::endl;  }  return 0;  } |

Этот код использует std::priority\_queue для выбора вершины с минимальным расстоянием, что делает реализацию эффективной. Граф представлен как список смежности, где каждая вершина хранит вектор пар (сосед, вес ребра).

## 3. Контейнеры STL в C++17

### 3.1. Роль STL в программировании

Стандартная библиотека шаблонов (STL) — это набор инструментов, предоставляемых C++ для работы с данными. Контейнеры STL — это классы, которые хранят и управляют набором элементов, таких как списки, массивы или словари. Они оптимизированы для различных задач и обеспечивают высокую производительность благодаря встроенным алгоритмам и структурам данных.

### 3.2. Категории контейнеров

1. **Последовательные контейнеры**
   * std::vector: динамический массив с быстрым доступом по индексу (O(1)).
   * std::list: двусвязный список для быстрой вставки и удаления (O(1)).
   * std::deque: двусторонняя очередь с эффективной работой на концах.
2. **Ассоциативные контейнеры**
   * std::set: упорядоченное множество уникальных элементов.
   * std::map: упорядоченный словарь “ключ-значение”.
3. **Неупорядоченные ассоциативные контейнеры**
   * std::unordered\_set: хеш-множество.
   * std::unordered\_map: хеш-таблица.

### 3.3. Нововведения C++17

### 3.3.1. std::string\_view

std::string\_view — это легковесный класс, который предоставляет доступ к строке без её копирования. Это особенно полезно при передаче строк в функции, где копирование std::string было бы избыточным.

**Пример кода на C++:**

|  |
| --- |
| void print(std::string\_view sv) {  std::cout << sv << std::endl;  }  int main() {  std::string s = "Hello, C++17!";  print(s); // Без копирования  return 0;  } |

**Псевдокод:**

функция print(sv):  
 вывести sv  
  
главная\_функция:  
 s = "Hello, C++17!"  
 print(s)

### 3.3.2. std::optional

std::optional позволяет безопасно работать с значениями, которые могут отсутствовать, заменяя использование указателей или “магических” значений (например, -1).

**Пример кода на C++:**

|  |
| --- |
| std::optional<int> find(int value, const std::vector<int>& vec) {  **for** (int v : vec) {  **if** (v == value) **return** v;  }  **return** std::nullopt; }  int main() {  std::vector<int> vec = {1, 2, 3};  **auto** result = find(4, vec);  **if** (result) {  std::cout << "Found: " << \*result << std::endl;  } **else** {  std::cout << "Not found" << std::endl;  }  **return** 0; } |

**Псевдокод:**

функция find(value, vec):  
 для каждого v в vec:  
 если v == value:  
 вернуть v  
 вернуть null  
  
главная\_функция:  
 vec = [1, 2, 3]  
 result = find(4, vec)  
 если result:  
 вывести "Found: ", result  
 иначе:  
 вывести "Not found"

## Заключение

Оптимизаторы, маршрутизаторы сложных структур данных и контейнеры STL в C++17 представляют собой мощный инструментарий, который позволяет разработчикам создавать высокопроизводительные, надёжные и удобные в использовании программы. Каждый из этих компонентов выполняет свою уникальную роль, но их взаимодействие создаёт синергетический эффект, который особенно важен в современных условиях, где требования к скорости и эффективности программного обеспечения постоянно растут.

Компиляторные оптимизации, такие как инлайн-подстановка функций, удаление мёртвого кода, константное свёртывание и оптимизация циклов, позволяют автоматически улучшать производительность кода, минимизируя усилия разработчика. Эти техники особенно полезны в массовых проектах, где ручная оптимизация каждого участка кода была бы непрактичной. Например, в игровых движках, где требуется обработка миллионов операций в секунду, компиляторные оптимизации могут сократить время рендеринга кадра на миллисекунды, что критично для обеспечения плавности изображения. Однако автоматические методы не всегда достаточны, и здесь на помощь приходят ручные оптимизации. Выбор эффективных алгоритмов, минимизация копирования данных и кэширование результатов позволяют разработчикам “выжать” максимум из аппаратных ресурсов, особенно в задачах с ограниченными вычислительными мощностями, таких как встраиваемые системы.

Маршрутизаторы сложных структур данных, такие как алгоритм Дейкстры, демонстрируют, как можно эффективно управлять данными в графах, деревьях и хеш-таблицах. Эти алгоритмы находят применение в самых разных областях: от маршрутизации сетевых пакетов в интернете до планирования маршрутов в системах GPS-навигации. Реализация таких алгоритмов в C++ с использованием STL, как показано в примере с алгоритмом Дейкстры, упрощает разработку благодаря готовым контейнерам, таким как std::vector и std::priority\_queue. Это позволяет сосредоточиться на логике задачи, а не на низкоуровневых деталях управления памятью или структуры данных.

Контейнеры STL в C++17, включая новые инструменты вроде std::string\_view и std::optional, расширяют возможности языка, делая код более выразительным и безопасным. Например, std::string\_view устраняет необходимость копирования строк при передаче их в функции, что может существенно ускорить обработку текстовых данных в приложениях, таких как парсеры или веб-серверы. std::optional же предоставляет элегантное решение для работы с необязательными значениями, что особенно полезно в системах, где данные могут быть недоступны, например, при запросах к базе данных или внешним API.

Взаимодействие этих трёх компонентов — оптимизаторов, маршрутизаторов и контейнеров — создаёт основу для разработки современных приложений. Например, в задаче построения маршрута в реальном времени (как в навигаторах) алгоритм Дейкстры может использовать std::unordered\_map для хранения графа дорог, компилятор оптимизирует доступ к данным через инлайн-подстановку, а разработчик вручную кэширует часто используемые участки маршрута для ускорения повторных вычислений. Такой подход демонстрирует, как теоретические знания о языке и его инструментах применяются на практике.

В заключение можно сказать, что освоение оптимизаторов, маршрутизаторов и контейнеров STL в C++17 — это не просто технический навык, а способ мышления, который позволяет разработчикам находить оптимальные решения для сложных задач. Эти инструменты дают возможность создавать программы, которые не только быстро работают, но и легко поддерживаются и масштабируются. В условиях стремительного развития технологий, где каждая миллисекунда производительности имеет значение, такие знания становятся конкурентным преимуществом для любого программиста.