**Белорусский государственный технологический университет**

**Факультет информационных технологий**

**Кафедра программной инженерии**

**Реферат**

По дисциплине «Основы алгоритмизации и программирования»

На тему «Оценка быстродействия циклических конструкций»

Выполнил:

Студент 1 курса 10 группы ПИ

Макаревич Кирилл Витальевич

Проверил: Белодед Николай Иванович

2024, Минск

Оглавление

[**Введение** 3](#_Toc182436448)

[**Применение конструкций циклов (кратко)** 3](#_Toc182436449)

[**Сравнение производительности различных конструкций циклов** 4](#_Toc182436450)

[**Результат работы программы** 8](#_Toc182436451)

[**Оправданность использования go to** 10](#_Toc182436452)

[**Заключение** 12](#_Toc182436453)

# **Введение**

Циклы играют ключевую роль в программировании, так как позволяют многократно выполнять блоки кода, упрощая решение задач, связанных с повторяющимися операциями. Существуют различные типы циклических конструкций, каждая из которых имеет свои особенности, достоинства и ограничения. Основные конструкции циклов, такие как for, while, do-while, for-each, а также оператор goto, могут существенно влиять на производительность программы и удобство работы с кодом.

Цель:

Провести сравнительный анализ различных конструкций циклов, выделить их оптимальные области применения, а также рассмотреть их влияние на скорость выполнения программы и структуру кода.

Задачи:

1. Описать синтаксис и основные особенности циклов for, while, do-while, for-each и оператора goto.
2. Проанализировать примеры использования каждой конструкции для различных типов задач.
3. Сравнить производительность циклических конструкций и определить наиболее быстрые и удобные для использования в различных ситуациях.
4. Выделить плюсы использования оператора goto в сравнении с другими конструкциями и обосновать его целесообразность.

# **Применение конструкций циклов (кратко)**

**Цикл for**: Используется для задач с фиксированным количеством повторений, когда известны начальные и конечные значения. Идеален для работы с индексами, например, при переборе элементов массива или выполнения операций заданное количество раз.

**Цикл while**: Подходит для ситуаций, когда количество повторений заранее неизвестно, но требуется выполнение до тех пор, пока выполняется определённое условие. Часто используется для проверки данных, ожидания ввода от пользователя или других условий, которые могут измениться со временем.

**Цикл do-while**: Применяется в ситуациях, когда нужно выполнить цикл хотя бы один раз перед проверкой условия. Удобен для меню, циклов ввода данных, когда требуется хотя бы одно выполнение перед проверкой на продолжение.

**Цикл for-each**: Используется для перебора элементов контейнеров (например, массивов, списков), когда не требуется управление индексами. Упрощает код, делая его более читаемым, особенно при работе с массивами и контейнерами данных.

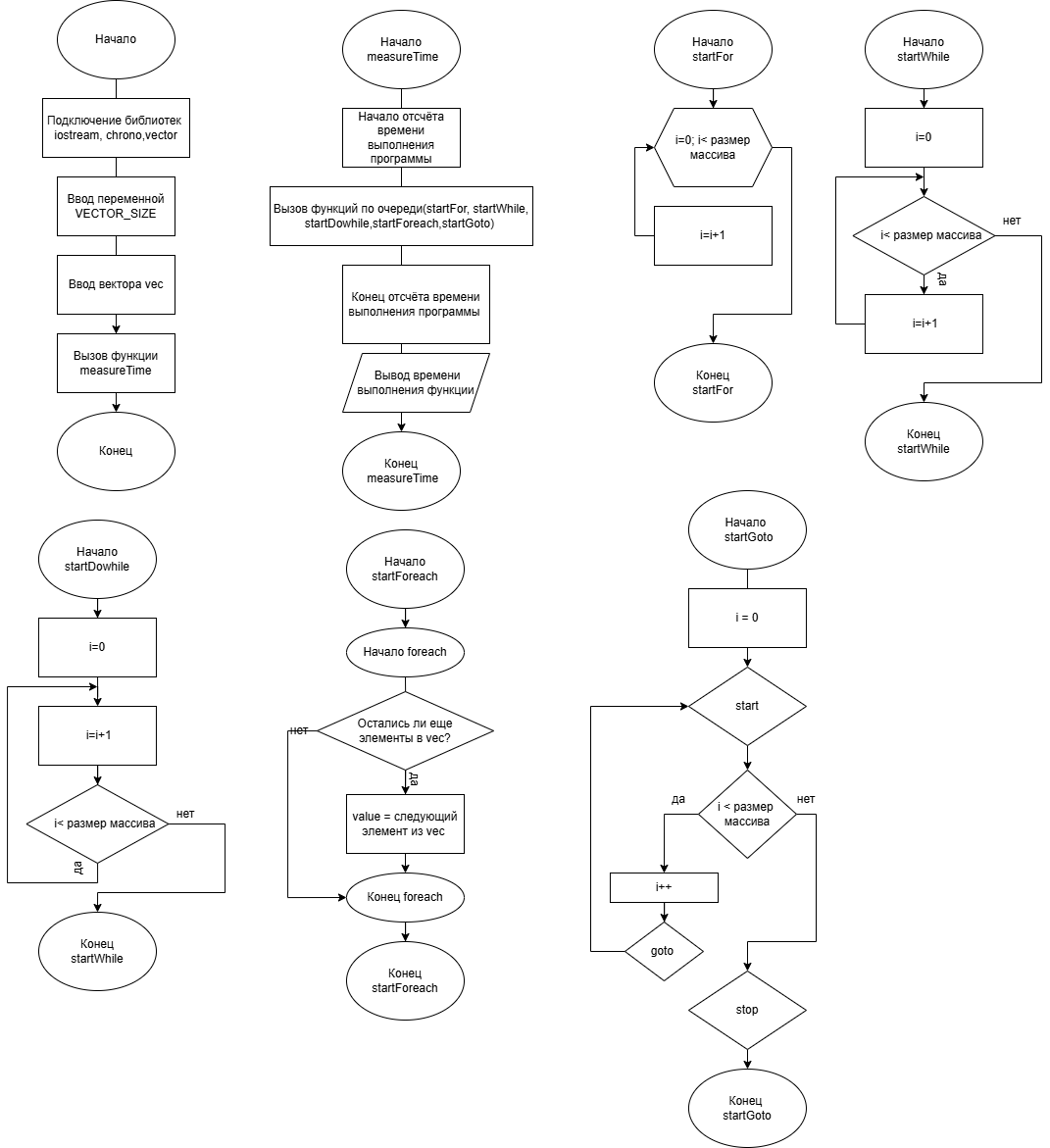
**Оператор goto**: Обычно применяется в редких случаях, например, для выхода из нескольких вложенных циклов или обработки ошибок, но из-за сложности в понимании кода его использование не рекомендуется. Далее расскажу про него более подробно.

# **Сравнение производительности различных конструкций циклов**

Код программы:

|  |
| --- |
| #include <iostream>  #include <chrono>  #include <vector>  const int VECTOR\_SIZE = 1000000000;  void startFor(const std::vector<int>& vec) {  for (size\_t i = 0; i < vec.size(); i++) {  }  }  void startWhile(const std::vector<int>& vec) {  size\_t i = 0;  while (i < vec.size()) {  i++;  }  }  void startDowhile(const std::vector<int>& vec) {  size\_t i = 0;  do {  i++;  } while (i < vec.size());  }  void startForeach(const std::vector<int>& vec) {  for (int value : vec) {  }  }  void startGoto(const std::vector<int>& vec) {  size\_t i = 0;  start:  if (i < vec.size()) {  i++;  goto start;  }  }  void measureTime(void (\*loopFunc)(const std::vector<int>&), const std::vector<int>& vec, const std::string& loopName) {  auto begin = std::chrono::steady\_clock::now();  loopFunc(vec);  auto end = std::chrono::steady\_clock::now();  auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin);  std::cout << loopName << " time: " << elapsed\_ms.count() << " ms\n";  }  int main() {  std::vector<int> vec(VECTOR\_SIZE, 1);  measureTime(startFor, vec, "for loop");  measureTime(startWhile, vec, "while loop");  measureTime(startDowhile, vec, "do while loop");  measureTime(startForeach, vec, "foreach loop");  measureTime(startGoto, vec, "goto loop");  return 0;  } |

Блок-схема



Псевдокод:

НАЧАЛО

ПОДКЛЮЧИТЬ БИБЛИОТЕКИ:

<iostream> — для работы с вводом-выводом

<chrono> — для измерения времени выполнения

<vector> — для использования контейнера вектор

ИНИЦИАЛИЗИРОВАТЬ КОНСТАНТУ:

VECTOR\_SIZE = 1000000000 (размер вектора)

СОЗДАТЬ И ЗАПОЛНИТЬ ВЕКТОР vec:

vec = вектор из VECTOR\_SIZE элементов, заполненных значением 1

ВЫЗВАТЬ ФУНКЦИЮ ДЛЯ ПОДСЧЕТА ВРЕМЕНИ ВЫПОЛНЕНИЯ КАЖДОГО ЦИКЛА:

Вызвать measureTime(startFor, vec, "for loop") — измеряет и выводит время выполнения цикла for

Вызвать measureTime(startWhile, vec, "while loop") — измеряет и выводит время выполнения цикла while

Вызвать measureTime(startDowhile, vec, "do while loop") — измеряет и выводит время выполнения цикла do-while

Вызвать measureTime(startForeach, vec, "foreach loop") — измеряет и выводит время выполнения цикла for-each

Вызвать measureTime(startGoto, vec, "goto loop") — измеряет и выводит время выполнения цикла goto

КОНЕЦ

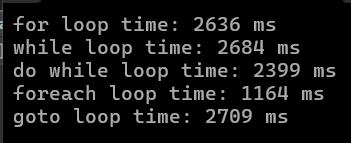
Словесно-формульное описание:

1. Начало программы
2. Подключение библиотек (<iostream>, <chrono>, <vector>)
3. Инициализация константы VECTOR\_SIZE — задается размер вектора (1 миллиард элементов).
4. Создание и заполнение вектора vec — создается вектор с VECTOR\_SIZE элементами, инициализированными значением 1.
5. Вызов функции для измерения времени выполнения каждого типа цикла — для каждого типа цикла (for, while, do-while, for-each, goto):
6. Запускается функция measureTime, которая:
7. Начинает отсчет времени,
8. Выполняет заданный цикл,
9. Фиксирует время окончания,
10. Выводит на экран время выполнения.
11. Завершение программы

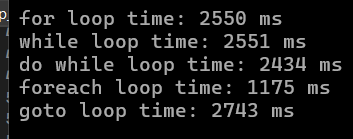
# **Результат работы программы**

Вывод результата (3 раза):

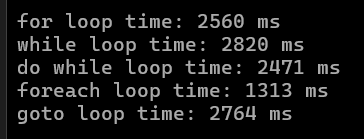
1:



2:



3:



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тип цикла | Измерение 1 (мс) | Измерение 2 (мс) | Измерение 3 (мс) | Сумма (мс) | Среднее время (мс) |
| for | 2636 | 2550 | 2560 | 7746 | 2582 |
| while | 2684 | 2551 | 2820 | 7955 | 2652 |
| do while | 2399 | 2434 | 2471 | 7304 | 2435 |
| foreach | 1164 | 1175 | 1313 | 3652 | 1217 |
| goto | 2709 | 2743 | 2764 | 8216 | 2739 |

На основе измеренных средних значений времени выполнения различных типов циклов можно сделать следующие выводы:

1. **Цикл foreach**: Цикл foreach продемонстрировал наименьшее среднее время выполнения — **1217 мс**. В отличие от обычных циклов for и while, цикл foreach не требует явного управления индексами или условиями завершения. Он предоставляет удобный и эффективный способ перебора элементов коллекций. Благодаря своей простоте и оптимизации в компиляторах, foreach оказался самым быстрым из всех тестируемых циклов. Его использование позволяет улучшить читаемость и уменьшить вероятность ошибок при переборе элементов, что делает его отличным выбором для операций с коллекциями данных.
2. **Циклы for, while, и do-while**: Средние времена выполнения этих циклов варьируются, но все они показали схожие результаты. Цикл for имеет среднее время выполнения **2582 мс**, цикл while — **2652 мс**, а цикл do-while — **2435 мс**. Эти циклы можно считать наиболее универсальными и часто используемыми. Они могут быть предпочтительны, когда необходимо явное управление индексами или когда количество итераций заранее неизвестно. В случае с циклом do-while он выполняется хотя бы один раз независимо от условия, что может быть полезно в некоторых сценариях, таких как меню или диалоговые окна. Однако, несмотря на схожесть в результатах, все эти циклы уступают по производительности циклу foreach, так как требуют больше операций для контроля условий завершения и индексов.
3. **Цикл goto**: Цикл с использованием оператора goto показал самое высокое среднее время выполнения — **2739 мс**. Это может быть связано с тем, что оператор goto нарушает нормальную структуру потока программы, что затрудняет оптимизацию компилятором. Каждое использование goto требует дополнительных переходов в коде, что делает его менее эффективным по сравнению с другими циклами.

# **Оправданность использования go to**

1. **Исторический контекст**:
   * goto и команды перехода являются базовыми элементами в низкоуровневом программировании, начиная с комбинационных схем и ассемблера. Автор напоминает, что переходы (в том числе условные и безусловные) используются во всех компьютерах и на языках низкого уровня (например, ассемблер).
2. **Сравнение кода с goto и без него**:
   * В некоторых случаях goto делает код гораздо более лаконичным и читаемым, чем при замене на флаги и вложенные условия. Без goto код становится запутанным и сложным для понимания, отладки и поддержки.
3. **Неявное использование переходов**:
   * Даже если программист избегает goto, многие операторы в C и C++ (такие как for, while, switch, break, continue, return, а также try и throw) используют переходы. Эти команды фактически представляют собой скрытую форму goto, хотя они считаются более «безопасными».
4. **Практические применения goto**:
   * Обработка ошибок: В языке C (в отличие от C++) нет исключений, поэтому goto часто используется для перехода к блокам обработки ошибок.

|  |
| --- |
| #include <stdio.h>  int open\_files() {  FILE\* file1 = fopen("file1.txt", "r");  if (!file1) goto cleanup;  FILE\* file2 = fopen("file2.txt", "r");  if (!file2) goto cleanup\_file1;  // Основной код работы с файлами  fclose(file2);  cleanup\_file1:  fclose(file1);  cleanup:  return -1;  } |

* + Выход из вложенных циклов: Когда нужно выйти из нескольких вложенных циклов, использование goto может быть более эффективным и понятным, чем вложенные условия или дополнительные флаги.

|  |
| --- |
| #include <iostream>  int main() {  for (int i = 0; i < 10; ++i) {  for (int j = 0; j < 10; ++j) {  if (i + j > 15) {  goto exit\_loops;  }  std::cout << i << ", " << j << std::endl;  }  }  exit\_loops:  std::cout << "Вышли из вложенных циклов." << std::endl;  return 0;  } |

* + Автоматически сгенерированный код: В коде, созданном автоматически (например, в лексических и синтаксических анализаторах), goto используется часто, так как это позволяет сократить объем кода и улучшить его производительность.
  + Единый выход из функции: В некоторых случаях удобно выйти из функции через одну точку (например, для освобождения ресурсов), и goto делает такой подход более читаемым.

|  |
| --- |
| #include <stdlib.h>  int process\_data() {  int\* data = (int\*)malloc(100 \* sizeof(int));  if (!data) return -1;  // Код обработки данных  if (some\_error\_occurred) goto cleanup;  // Дальнейшая обработка данных  if (another\_error\_occurred) goto cleanup;  cleanup:  free(data);  return -1;  } |

1. **Заключение про goto**:

goto — это инструмент, который нужно использовать осознанно, как нож или любое другое средство, обладающее потенциальными рисками. goto имеет свои недостатки, но иногда это оправданный и полезный инструмент, если применять его аккуратно и в подходящих ситуациях. что важно писать код с умом и не ограничивать себя общими правилами, если они приводят к усложнению кода без значимых преимуществ.

Шутка-минутка



# **Заключение**

Циклы являются неотъемлемой частью программирования, и выбор подходящей конструкции для конкретной задачи может существенно повлиять на производительность, читаемость и поддержку кода. В ходе анализа различных циклических конструкций (for, while, do-while, for-each, и goto) были рассмотрены их особенности и области применения.

Цикл foreach продемонстрировал наименьшее время выполнения, что делает его предпочтительным выбором при переборе элементов коллекций, так как он упрощает код и уменьшает вероятность ошибок. Циклы for, while и do-while показали схожие результаты, что делает их универсальными для задач с заранее известным количеством итераций или для выполнения хотя бы одного шага в цикле. В свою очередь, цикл с использованием оператора goto показал худшие результаты по времени выполнения, что связано с тем, что переходы с использованием этого оператора нарушают нормальную структуру потока программы, усложняя оптимизацию компилятора.

Тем не менее, оператор goto имеет свои обоснованные применения, такие как обработка ошибок, выход из вложенных циклов и улучшение читаемости в некоторых специфичных случаях, где другие подходы могут привести к значительному усложнению кода. Однако его использование должно быть осознанным и аккуратным, поскольку оно может сделать код менее понятным и сложным для отладки.

В заключение можно сказать, что выбор конструкции цикла или оператора зависит от конкретных задач. В большинстве случаев следует отдавать предпочтение более читаемым и понятным решениям, таким как foreach, но в особых случаях использование goto может быть оправданным, если это помогает улучшить структуру программы или повысить её производительность.