**Белорусский государственный технологический университет**

**Факультет информационных технологий**

**Кафедра программной инженерии**

**Реферат “** **Функторы и лямбда-функции: отличия и подобия”**

По дисциплине «Основы алгоритмизации и программирования»

Выполнил:

Студент 1 курса 10 группы ПИ

Макаревич Кирилл Витальевич

Проверил: Белодед Николай Иванович

Оглавление

[**Введение** 2](#_Toc195055407)

[**Функторы в C++** 2](#_Toc195055408)

[Определение и назначение: 3](#_Toc195055409)

[Реализация функтора: 3](#_Toc195055410)

[Применение в STL 4](#_Toc195055411)

[**Лямбда-функции в C++** 5](#_Toc195055412)

[Определение и назначение 5](#_Toc195055413)

[Синтаксис лямбда-функции 6](#_Toc195055414)

[Реализация лямбда-функции 6](#_Toc195055415)

[Применение в STL 7](#_Toc195055416)

[**Сравнение функторов и лямбда-функций** 7](#_Toc195055417)

[Основные сходства 7](#_Toc195055418)

[Основные отличия 8](#_Toc195055419)

[Пример совместного использования 9](#_Toc195055420)

[**Заключение** 9](#_Toc195055421)

## 

## **Введение**

Язык программирования C++ — это мощный инструмент, который сочетает в себе возможности низкоуровневого управления памятью с высокоуровневыми абстракциями, такими как объектно-ориентированное и функциональное программирование. Среди множества конструкций, предоставляемых C++, особое внимание заслуживают функторы и лямбда-функции. Эти механизмы играют ключевую роль в создании гибких, переиспользуемых и выразительных программ, особенно когда речь идёт о работе со стандартной библиотекой шаблонов (STL). STL, включающая такие компоненты, как контейнеры (std::vector, std::map) и алгоритмы (std::sort, std::transform), активно использует функторы и лямбда-функции для настройки поведения своих методов.

Функторы появились в C++ ещё на ранних этапах его развития, когда язык только начинал формироваться как расширение C с поддержкой объектно-ориентированного подхода. Они стали естественным способом внедрения функционального стиля в язык, где традиционные функции были ограничены отсутствием состояния. С другой стороны, лямбда-функции — более современное дополнение, введённое в стандарте C++11. Их появление ознаменовало новый этап в эволюции C++, сделав язык более удобным для функционального программирования и приблизив его к таким языкам, как Python или JavaScript, где анонимные функции являются неотъемлемой частью синтаксиса. Лямбда-функции упростили написание кода, устранив необходимость создавать отдельные классы или функции для одноразовых задач, и добавили гибкость благодаря механизму захвата переменных.

Цель данного реферата — провести глубокий анализ функторов и лямбда-функций, раскрыть их отличия и сходства, а также показать, как они применяются на практике в современных программах на C++. Для достижения этой цели будут рассмотрены теоретические аспекты обеих конструкций, приведены примеры кода с псевдокодом для лучшего понимания их работы, а также составлены таблицы сравнения, которые наглядно иллюстрируют их особенности. Такой подход позволит не только понять технические детали, но и оценить, в каких ситуациях лучше использовать ту или иную конструкцию.

## **Функторы в C++**

### Определение и назначение:

Функтор (или функциональный объект) — это объект класса или структуры, для которого перегружен оператор вызова функции operator(). Благодаря этой перегрузке объект можно использовать так же, как обычную функцию, вызывая его с помощью круглых скобок и передавая аргументы. Функторы являются важной частью экосистемы C++, особенно в контексте стандартной библиотеки шаблонов (STL), где они часто применяются для передачи пользовательской логики в алгоритмы, такие как std::sort, std::find\_if или std::transform. Например, если стандартное поведение алгоритма сортировки (по возрастанию) не подходит, функтор позволяет задать собственный критерий сравнения, например, сортировку по убыванию или по модулю.

Одним из главных преимуществ функторов по сравнению с обычными функциями является их способность хранить состояние. Обычная функция в C++ не имеет встроенного механизма для сохранения данных между вызовами, если не использовать глобальные переменные, что считается плохой практикой. Функтор же, как объект класса, может содержать поля, которые инициализируются при создании объекта и используются при каждом вызове. Это делает функторы мощным инструментом в ситуациях, где нужно сохранять промежуточные результаты, накапливать данные или передавать дополнительные параметры без необходимости явно указывать их в каждом вызове. Например, функтор может подсчитывать количество своих вызовов или настраиваться на определённое поведение через конструктор.

Функторы также обладают высокой степенью переиспользуемости. Один и тот же класс функтора можно использовать в разных частях программы, передавая его экземпляры в различные алгоритмы или функции. Это делает их удобными для модульного проектирования, где требуется единообразная обработка данных в нескольких местах.

### Реализация функтора:

Чтобы лучше понять, как работают функторы, рассмотрим пример реализации простого функтора, который выполняет операцию сложения заданного числа с переданным аргументом.

|  |
| --- |
| struct Adder {  int value;  Adder(int v) : value(v) {}  int operator()(int x) const {  return x + value;  }  };  int main() {  Adder add5(5); // Создаём функтор с состоянием value = 5  std::cout << add5(10) << std::endl; // Вывод: 15  return 0;  } |

***Псевдокод:***

структура Adder:

поле value

конструктор Adder(v):

value = v

функция operator()(x):

вернуть x + value

главная\_функция:

add5 = Adder(5)

вывести add5(10)

В этом примере объект `add5` ведёт себя как функция, принимающая аргумент `x` и возвращающая сумму `x + value`. Состояние `value` задаётся при создании объекта и остаётся неизменным благодаря модификатору `const`.

**Расширение примера: изменяемое состояние**

Теперь рассмотрим случай, когда функтор может изменять своё состояние, например, подсчитывать количество вызовов.

|  |
| --- |
| struct Counter {  int count; // Счётчик вызовов  Counter() : count(0) {} // Инициализация счётчика нулем  int operator()(int x) { // Без const, так как изменяем состояние  return x + (++count); // Увеличиваем счётчик перед сложением  }  };  int main() {  Counter counter; // Создаём экземпляр функтора  std::cout << counter(10) << std::endl; // Вывод: 11 (count = 1)  std::cout << counter(10) << std::endl; // Вывод: 12 (count = 2)  std::cout << counter(10) << std::endl; // Вывод: 13 (count = 3)  return 0;  } |

***Псевдокод:***

структура Counter:

поле count

конструктор Counter():

count = 0

функция operator()(x):

count = count + 1

вернуть x + count

главная\_функция:

counter = Counter()

вывести counter(10)

вывести counter(10)

вывести counter(10)

Этот пример подчёркивает гибкость функторов: они могут быть как неизменяемыми (как в первом случае), так и динамически изменяемыми (как во втором), в зависимости от требований задачи.

Применение в STL

Функторы часто используются в алгоритмах STL. Например, для сортировки вектора по убыванию можно создать функтор сравнения.

|  |
| --- |
| struct Greater {  bool operator()(int a, int b) const {  return a > b;  }  };  int main() {  std::vector<int> vec = { 3, 1, 4, 1, 5 };  std::sort(vec.begin(), vec.end(), Greater());  for (int x : vec) {  std::cout << x << " "; // Вывод: 5 4 3 1 1  }  std::cout << std::endl;  return 0;  } |

*Псевдокод:*

структура Greater:

функция operator()(a, b):

вернуть a > b

главная\_функция:

vec = [3, 1, 4, 1, 5]

сортировать(vec, Greater)

для каждого x в vec:

вывести x

Здесь функтор `Greater` передаётся в `std::sort` как критерий сортировки, заменяя стандартное поведение (сортировка по возрастанию).

**Лямбда-функции в C++**

Определение и назначение

Лямбда-функции (или лямбда-выражения) были введены в стандарт C++11 как значительное новшество, расширяющее возможности языка в области функционального программирования. Они представляют собой способ создания анонимных функций — то есть функций без имени, которые определяются непосредственно в месте их использования. Это позволяет разработчикам писать компактный и выразительный код, избегая необходимости создавать отдельные именованные функции или классы для задач, которые используются лишь однажды. Появление лямбда-функций сделало C++ более современным языком, приблизив его к таким языкам, как Python, Java (с версии 8) или JavaScript, где анонимные функции являются стандартной практикой.

Основное назначение лямбда-функций — упрощение кода и повышение его читаемости в ситуациях, где требуется передать небольшую функциональность в качестве аргумента другой функции или алгоритма. Они особенно удобны в сочетании со стандартной библиотекой шаблонов (STL), где часто возникает необходимость настройки поведения алгоритмов, таких как std::sort, std::for\_each, std::find\_if или std::transform. Например, вместо написания отдельной функции или функтора для проверки условия в std::find\_if, можно использовать лямбда-выражение прямо в вызове алгоритма, что сокращает объём кода и делает его более интуитивным.

Одной из ключевых особенностей лямбда-функций является их способность захватывать переменные из окружающей области видимости. Это позволяет лямбда-выражениям работать с данными, определёнными вне их тела, без необходимости передавать эти данные через параметры. Такой механизм делает лямбда-функции гибкими и мощными, так как они могут адаптироваться к контексту, в котором используются, сохраняя при этом лаконичность. Например, лямбда-функция может захватить локальную переменную и использовать её для вычислений, что особенно полезно в обработке данных или реализации обратных вызовов (callbacks).

Лямбда-функции также поддерживаются современными возможностями C++, такими как автоматический вывод типов (auto), что упрощает их хранение и повторное использование. В отличие от традиционных функций, которые требуют явного объявления в отдельной части кода, лямбда-выражения интегрируются непосредственно в поток программы, что делает их идеальными для быстрого прототипирования и одноразовых задач. Кроме того, компиляторы часто оптимизируют лямбда-функции путём их инлайн-подстановки, что может улучшить производительность по сравнению с обычными функциями или даже функторами в некоторых случаях.

### Синтаксис лямбда-функции

Синтаксис лямбда-выражения в C++ выглядит следующим образом:  
**[захват](параметры) -> возвращаемый\_тип { тело }**

Разберём каждый элемент подробнее:

1. **Захват ([захват])**  
   Захват определяет, какие переменные из окружающей области видимости доступны внутри лямбда-функции и как они будут использоваться. Этот механизм является уникальной особенностью лямбда-выражений и отличает их от других способов определения функций в C++. Существует несколько вариантов захвата:
   * [] — пустой захват. Лямбда-функция не имеет доступа к внешним переменным и работает только с переданными параметрами. Это делает её похожей на обычную функцию с изолированной областью видимости.
   * [=] — захват всех переменных по значению. Копии всех доступных внешних переменных передаются в лямбда-функцию. Это удобно, когда нужно использовать неизменяемые данные из контекста.
   * [&] — захват всех переменных по ссылке. Лямбда-функция получает прямой доступ к внешним переменным, что позволяет изменять их внутри тела лямбда-выражения.
   * [var] или [&var] — захват конкретной переменной по значению или по ссылке. Это даёт точный контроль над тем, какие данные используются.
   * Смешанный захват, например [=,&var], позволяет комбинировать захват по значению для всех переменных и по ссылке для конкретной.

**Пример захвата:**

|  |
| --- |
| int a = 5;  auto lambda = [a](int x) { return x + a; }; // Захват a по значению |

1. **Параметры ((параметры))**  
   Список параметров указывается в круглых скобках и работает так же, как у обычных функций. Параметры определяют, какие аргументы принимает лямбда-функция при вызове. Если аргументы не нужны, скобки могут быть пустыми: (). Например:

* (int x) — принимает один целочисленный аргумент.
* (int x, int y) — принимает два аргумента.
* () — не принимает аргументов.

Параметры могут использовать спецификаторы, такие как const или &, для управления способом передачи данных.

1. **Возвращаемый тип (-> возвращаемый\_тип)**  
   Возвращаемый тип указывается после стрелки -> и является опциональным. Если тип не указан, компилятор автоматически выводит его на основе выражения в теле лямбда-функции (обычно по оператору return). Например:

* -> int — явно указывает, что лямбда возвращает целое число.
* Без -> — компилятор сам определяет тип, что упрощает код в простых случаях.

Пример с явным типом:

|  |
| --- |
| auto lambda = [](int x) -> double { return x + 0.5; }; |

1. **Тело ({ тело })**  
   Тело лямбда-функции содержит код, который выполняется при её вызове. Оно заключается в фигурные скобки и может включать несколько операторов, переменные и даже вызовы других функций. В простых случаях тело может быть одной строкой, например:
   * { return x + 1; } — возвращает результат выражения.
   * { std::cout << x; return x \* 2; } — содержит несколько действий.

Реализация лямбда-функции

Рассмотрим аналогичный пример сложения с использованием лямбда-функции.

|  |
| --- |
| int main() {  int value = 5;  auto add = [value](int x) { return x + value; };  std::cout << add(10) << std::endl; // Вывод: 15  return 0;  } |

*Псевдокод:*

главная\_функция:

value = 5

add = лямбда(x):

вернуть x + value

вывести add(10)

Здесь лямбда-функция захватывает переменную `value` по значению и использует её для сложения с аргументом `x`.

Применение в STL

Лямбда-функции часто заменяют функторы в алгоритмах STL благодаря своей компактности.

|  |
| --- |
| int main() {  std::vector<int> vec = { 3, 1, 4, 1, 5 };  std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b; });  for (int x : vec) {  std::cout << x << " "; // Вывод: 5 4 3 1 1  }  std::cout << std::endl;  return 0;  } |

*Псевдокод:*

главная\_функция:

vec = [3, 1, 4, 1, 5]

сортировать(vec, лямбда(a, b): вернуть a > b)

для каждого x в vec:

вывести x

Лямбда-выражение задаётся прямо в вызове `std::sort`, что делает код короче и понятнее.

**Сравнение функторов и лямбда-функций**

Основные сходства

Функторы и лямбда-функции имеют много общего, так как оба механизма предназначены для создания объектов, которые можно вызывать как функции.

**Сходства функторов и лямбда-функций**

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **Функторы** | **Лямбда-функции** |
| Вызываемость | Да, через operator() | Да, как анонимная функция |
| Использование в STL | Поддерживается | Поддерживается |
| Хранение состояния | Да, через поля класса | Да, через захват переменных |
| Тип | Объект пользовательского типа | Объект скрытого типа |

Оба подхода позволяют передавать поведение в алгоритмы STL и поддерживают хранение состояния, что делает их взаимозаменяемыми в ряде случаев.

Основные отличия

Несмотря на сходства, между функторами и лямбда-функциями есть существенные различия, которые влияют на их применение.

**Отличия функторов и лямбда-функций**

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **Функторы** | **Лямбда-функции** |
| Определение | Требуется отдельный класс или структура | Определяются inline в месте использования |
| Читабельность | Может быть громоздким для простых задач | Компактны и выразительны |
| Гибкость захвата | Ограничена полями класса | Поддерживает захват по значению/ссылке |
| Повторное использование | Легко переиспользовать в разных местах | Обычно одноразовые, но можно сохранить |
| Инлайн-оптимизация | Зависит от компилятора | Часто оптимизируются лучше благодаря inline |
| Именование | Имеет явное имя (класс) | Анонимны, но можно присвоить имя через auto |

Пример совместного использования

Функторы и лямбда-функции можно комбинировать для решения сложных задач. Например, функтор может задавать базовое поведение, а лямбда-функция — настраивать его.

|  |
| --- |
| struct Counter {  int count = 0;  int operator()(int x) {  return ++count + x;  }  };  int main() {  std::vector<int> vec = { 1, 2, 3 };  Counter counter;  auto combined = [&counter](int x) { return counter(x) \* 2; };  std::transform(vec.begin(), vec.end(), vec.begin(), combined);  for (int x : vec) {  std::cout << x << " "; // Вывод: 4 6 8  }  std::cout << std::endl;  return 0;  } |

*Псевдокод:*

структура Counter:

count = 0

функция operator()(x):

count = count + 1

вернуть count + x

главная\_функция:

vec = [1, 2, 3]

counter = Counter

combined = лямбда(x): вернуть counter(x) \* 2

преобразовать(vec, combined)

для каждого x в vec:

вывести x

Здесь функтор `Counter` считает вызовы, а лямбда-функция удваивает результат, демонстрируя гибкость комбинирования.

**Заключение**

Функторы и лямбда-функции в C++ — это два мощных инструмента, которые решают схожие задачи, но имеют разные области применения. Функторы, как объекты с состоянием и явной структурой, идеально подходят для сложных, повторно используемых логик, таких как обработка данных в больших проектах. Лямбда-функции, благодаря своей компактности и гибкости захвата, лучше подходят для одноразовых задач и быстрого написания кода, особенно в сочетании с алгоритмами STL.

Сравнительные таблицы показывают, что выбор между функторами и лямбда-функциями зависит от контекста: для простоты и читаемости лучше использовать лямбда-выражения, а для сложных систем с долгосрочным переиспользованием — функторы. Их совместное применение, как в приведённом примере, позволяет сочетать преимущества обоих подходов, создавая гибкие и эффективные решения.

В современном C++ (особенно с учётом стандартов C++11 и выше) лямбда-функции часто вытесняют функторы благодаря удобству и оптимизациям компилятора, но знание обоих инструментов остаётся важным для профессионального разработчика. Понимание их отличий и подобий помогает выбирать оптимальный подход для каждой конкретной задачи, что делает код не только производительным, но и удобным для поддержки.