**Белорусский государственный технологический университет**

**Факультет информационных технологий**

**Кафедра программной инженерии**

**Реферат “** **Современные стандарты C++: C++11/14/17/20/23 – ключевые нововведения”**

По дисциплине «Основы алгоритмизации и программирования»

Выполнил:

Студент 1 курса 10 группы ПИ

Макаревич Кирилл Витальевич

Проверил: Белодед Николай Иванович

Оглавление

[**1. Введение** 2](#_Toc196933528)

[**2. C++11: Фундамент современных стандартов** 2](#_Toc196933529)

[**3. C++14: Уточнения и улучшения** 4](#_Toc196933530)

[**4. C++17: Углубление функциональности** 5](#_Toc196933531)

[**5. C++20: Новый уровень абстракции** 7](#_Toc196933532)

[**6. C++23: Перспективы и доработки** 9](#_Toc196933533)

[**7. Сравнительный анализ и влияние на разработку** 10](#_Toc196933534)

[**8. Заключение** 11](#_Toc196933535)

# **1. Введение**

Язык программирования C++, созданный Бьярне Страуструпом в 1980-х годах, остаётся одним из самых мощных и универсальных инструментов в арсенале разработчиков. Его применяют в самых разных областях: от системного программирования (операционные системы, драйверы) до разработки высокопроизводительных приложений, таких как игровые движки, финансовые системы и научные симуляции. Уникальность C++ заключается в сочетании низкоуровневого контроля над ресурсами с высокоуровневыми абстракциями, что делает его востребованным в условиях растущих требований к производительности и масштабируемости.

С 2011 года, с выходом стандарта C++11, язык переживает ренессанс. Регулярные обновления стандартов (C++11, C++14, C++17, C++20, C++23), разрабатываемые под эгидой международного комитета ISO/IEC, привносят инновации, которые делают C++ более безопасным, выразительным и удобным. Эти стандарты не только устраняют недостатки предыдущих версий, но и внедряют новые парадигмы, такие как модульность, асинхронное программирование и концепты. Они позволяют C++ конкурировать с современными языками, такими как Rust, Go и Python, сохраняя при этом производительность и гибкость.

# **2. C++11: Фундамент современных стандартов**

C++11, принятый в 2011 году, стал переломным моментом в истории языка, заложив основу для его модернизации. Этот стандарт сделал C++ более безопасным, выразительным и пригодным для многопоточных приложений, устранив многие неудобства C++98/03.

* **Автоматическое определение типов (auto и decltype)**: Ключевое слово auto позволяет компилятору самостоятельно определять тип переменной на основе её инициализатора, что упрощает код, особенно при работе с итераторами и шаблонами. decltype извлекает тип выражения, что полезно в метапрограммировании и шаблонном программировании.
* **Лямбда-выражения**: Анонимные функции вида [capture](params){body} упростили написание локальных обработчиков, интеграцию с алгоритмами STL и создание функционального стиля программирования.
* **Умные указатели**: std::shared\_ptr и std::unique\_ptr заменили сырые указатели, автоматизируя управление памятью и предотвращая утечки. std::weak\_ptr дополняет их, решая проблему циклических ссылок.
* **Range-based for циклы**: Синтаксис for (auto& x : container) сделал итерацию по контейнерам интуитивной и читаемой.
* **Многопоточность**: Введение std::thread, std::mutex, std::condition\_variable и std::atomic обеспечило встроенную поддержку параллельного программирования, устранив зависимость от платформозависимых библиотек, таких как POSIX Threads.
* **nullptr**: Новый указатель на null устранил неоднозначности, связанные с использованием 0 или NULL, улучшив типобезопасность.
* **Variadic templates**: Шаблоны с переменным числом аргументов расширили возможности метапрограммирования, позволив создавать гибкие функции, такие как std::tuple и улучшенные версии printf.
* **Инициализаторы списков**: Синтаксис std::vector<int> v = {1, 2, 3}; упростил инициализацию контейнеров и структур.

**Пример кода 1 (C++11)**: Использование auto, лямбда-выражений и std::shared\_ptr.

|  |
| --- |
| #include <iostream>  #include <vector>  #include <algorithm>  #include <memory>  int main() {  std::vector<int> vec = { 5, 2, 9, 1, 5 };  auto print = [](const auto& v) {  for (const auto& x : v) std::cout << x << " ";  std::cout << "\n";  };  print(vec);  std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b; });  print(vec);  auto ptr = std::make\_shared<std::vector<int>>(vec);  std::cout << "Shared ptr size: " << ptr->size() << "\n";  } |

**Псевдокод 1**:

|  |
| --- |
| создать вектор чисел {5, 2, 9, 1, 5}  определить функцию print:  для каждого элемента x в векторе:  вывести x  вывести новую строку  вызвать print для вектора  отсортировать вектор по убыванию, используя сравнение a > b  вызвать print для отсортированного вектора  создать shared\_ptr на копию вектора  вывести размер вектора через shared\_ptr |

**Пример кода 2 (C++11)**: Многопоточность с std::thread и std::mutex.

|  |
| --- |
| #include <iostream>  #include <thread>  #include <mutex>  std::mutex mtx;  int counter = 0;  void increment(int n) {  for (int i = 0; i < n; ++i) {  std::lock\_guard<std::mutex> lock(mtx);  ++counter;  }  }  int main() {  std::thread t1(increment, 1000);  std::thread t2(increment, 1000);  t1.join();  t2.join();  std::cout << "Final counter: " << counter << "\n";  } |

**Псевдокод 2**:

|  |
| --- |
| определить мьютекс mtx  определить глобальный счётчик counter = 0  определить функцию increment(n):  для i от 0 до n:  заблокировать мьютекс  увеличить counter  разблокировать мьютекс  создать два потока, каждый вызывает increment(1000)  дождаться завершения обоих потоков  вывести значение counter |

**Практическое применение**: C++11 широко используется в современных библиотеках, таких как Boost, и игровых движках, таких как Unreal Engine. Например, умные указатели и лямбда-выражения упрощают управление ресурсами и обработку событий в реальном времени, а многопоточность позволяет эффективно использовать многоядерные процессоры в симуляциях и рендеринге.

C++11 заложил фундамент для последующих стандартов, сделав язык более безопасным, выразительным и пригодным для параллельного программирования.

# **3. C++14: Уточнения и улучшения**

C++14, принятый в 2014 году, стал эволюционным обновлением, уточняющим и дополняющим возможности C++11. Он сфокусировался на упрощении синтаксиса и повышении удобства разработки.

* **Упрощённое использование auto**: Теперь auto можно использовать для возвращаемых типов функций без необходимости явного указания decltype, что упрощает написание шаблонных функций.
* **Лямбда-выражения с инициализаторами**: Поддержка захвата переменных с инициализацией (например, [x = expr]) сделала лямбды более гибкими для сложной логики.
* **Constexpr функции**: Расширение возможностей constexpr позволило выполнять более сложные вычисления на этапе компиляции, улучшая производительность и открывая новые возможности для метапрограммирования.
* **Стандартные литералы**: Пространство имён std::literals добавило удобные суффиксы, такие как "text"s для std::string и 42i для комплексных чисел.
* **Улучшения шаблонов**: Упрощённые правила вывода типов для шаблонов и поддержка decltype(auto) повысили гибкость шаблонного программирования.

**Пример кода 1 (C++14)**: Использование auto для возвращаемого типа и constexpr функции.

|  |
| --- |
| #include <iostream>  #include <string>  constexpr int factorial(int n) {  return n <= 1 ? 1 : n \* factorial(n - 1);  }  auto concatenate(const std::string& a, const std::string& b) {  using namespace std::literals;  return a + b + "!"s;  }  int main() {  std::cout << "Factorial(5) = " << factorial(5) << "\n";  std::cout << concatenate("Hello", "World") << "\n";  } |

**Псевдокод 1**:

|  |
| --- |
| определить константную функцию factorial(n):  если n <= 1, вернуть 1  иначе вернуть n \* factorial(n-1)  определить функцию concatenate(a, b):  вернуть a + b + "!"  вывести factorial(5)  вывести concatenate("Hello", "World") |

**Пример кода 2 (C++14)**: Лямбда-выражения с инициализаторами.

|  |
| --- |
| #include <iostream>  #include <vector>  #include <algorithm>  int main() {  std::vector<int> vec = { 1, 2, 3, 4, 5 };  int offset = 10;  std::for\_each(vec.begin(), vec.end(), [offset, counter = 0](int& x) mutable {  x += offset + counter++;  std::cout << x << " ";  });  std::cout << "\n";  } |

**Псевдокод 2**:

|  |
| --- |
| создать вектор чисел {1, 2, 3, 4, 5}  определить offset = 10  для каждого элемента x в векторе:  увеличить x на offset + counter  увеличить counter  вывести x  вывести новую строку |

**Практическое применение**: C++14 активно используется в проектах, где важна читаемость кода и оптимизация на этапе компиляции. Например, constexpr функции применяются в научных вычислениях для генерации таблиц констант, а улучшенные лямбда-выражения упрощают обработку данных в библиотеках обработки сигналов.

C++14 сделал язык более удобным, сохранив совместимость с C++11, и подготовил почву для более значительных изменений в C++17.

# **4. C++17: Углубление функциональности**

C++17, принятый в 2017 году, расширил возможности языка, добавив инструменты для работы с данными, файловыми системами и оптимизации производительности.

* **Структурированные привязки**: Синтаксис auto [a, b] = pair; упростил распаковку кортежей, структур и других составных типов.
* **Варианты**: Типы std::variant, std::optional и std::any предоставили безопасные альтернативы для работы с разнотипными данными, уменьшив необходимость в сырых указателях.
* **Fold expressions**: Сворачивание выражений для variadic templates упростило обработку параметров в шаблонах, сделав код более компактным.
* **Файловая система**: Библиотека std::filesystem добавила стандартные средства для работы с файлами и директориями, устранив зависимость от платформозависимых API.
* **Параллельные алгоритмы**: STL получила поддержку параллельного выполнения алгоритмов, таких как std::sort и std::transform, с использованием многопоточности.
* **Улучшения шаблонов**: Введение if constexpr позволило условно компилировать код в шаблонах, улучшая читаемость и производительность.

**Пример кода 1 (C++17)**: Структурированные привязки и std::optional.

|  |
| --- |
| #include <iostream>  #include <optional>  #include <tuple>  std::tuple<std::string, int> get\_info() {  return { "example.txt", 42 };  }  std::optional<int> safe\_divide(int a, int b) {  if (b == 0) return std::nullopt;  return a / b;  }  int main() {  auto [name, size] = get\_info();  std::cout << "File: " << name << ", Size: " << size << "\n";  if (auto result = safe\_divide(10, 2); result) {  std::cout << "Result: " << \*result << "\n";  }  else {  std::cout << "Division failed\n";  }  } |

**Псевдокод 1**:

|  |
| --- |
| определить функцию get\_info:  вернуть кортеж ("example.txt", 42)  определить функцию safe\_divide(a, b):  если b == 0, вернуть null  вернуть a / b  распаковать результат get\_info в name и size  вывести name и size  вызвать safe\_divide(10, 2)  если результат существует:  вывести результат  иначе:  вывести "Division failed" |

**Пример кода 2 (C++17)**: Использование std::filesystem и параллельных алгоритмов.

|  |
| --- |
| #include <iostream>  #include <filesystem>  #include <vector>  #include <algorithm>  #include <execution>  int main() {  std::vector<int> vec(1000000);  std::generate(vec.begin(), vec.end(), [n = 0]() mutable { return n++; });  std::sort(std::execution::par, vec.begin(), vec.end());  std::cout << "First 5 sorted elements: ";  for (int i = 0; i < 5; ++i) std::cout << vec[i] << " ";  std::cout << "\n";  for (const auto& entry : std::filesystem::directory\_iterator(".")) {  std::cout << entry.path() << "\n";  }  } |

**Псевдокод 2**:

|  |
| --- |
| создать вектор из 1000000 элементов  заполнить вектор числами от 0, увеличивая счётчик  параллельно отсортировать вектор  вывести первые 5 элементов  для каждого файла в текущей директории:  вывести путь к файлу |

**Практическое применение**: C++17 широко используется в кроссплатформенных приложениях. Например, std::filesystem упрощает работу с файлами в инструментах сборки, таких как CMake, а параллельные алгоритмы повышают производительность в обработке больших данных, например, в машинном обучении.

C++17 сделал язык более универсальным, предоставив мощные инструменты для работы с данными и файловыми системами.

# **5. C++20: Новый уровень абстракции**

C++20, принятый в 2020 году, стал одним из самых амбициозных стандартов, внедрив новые парадигмы, которые сделали C++ конкурентом высокоуровневых языков, сохраняя его производительность.

* **Модули**: Система модулей заменила традиционные заголовочные файлы, улучшив скорость компиляции, инкапсуляцию и читаемость кода.
* **Концепты**: Ключевое слово concept и модификатор requires позволили задавать ограничения на шаблоны, делая код более читаемым и предотвращая ошибки.
* **Ranges**: Библиотека std::ranges переосмыслила работу с последовательностями, упрощая алгоритмы и добавляя функциональный стиль.
* **Улучшенные лямбда-выражения**: Поддержка stateless лямбд, явных шаблонных параметров и захвата this сделала их ещё мощнее.
* **Корутины**: Новый механизм (co\_await, co\_yield, co\_return) упростил асинхронное программирование и создание генераторов.
* **Новые типы**: std::span для безопасной работы с непрерывными данными и улучшения в std::bitset.
* **Календарь и временные зоны**: Библиотека std::chrono получила поддержку дат, календарей и временных зон.

**Пример кода 1 (C++20)**: Использование концептов и ranges.

|  |
| --- |
| #include <iostream>  #include <vector>  #include <ranges>  #include <concepts>  template<typename T>  concept Numeric = std::integral<T> || std::floating\_point<T>;  template<Numeric T>  void print\_range(const std::vector<T>& vec) {  for (auto v : vec | std::views::filter([](T x) { return x > 0; })) {  std::cout << v << " ";  }  std::cout << "\n";  }  int main() {  std::vector<int> vec = { -1, 2, 3, -4, 5 };  print\_range(vec);  } |

**Псевдокод 1**:

|  |
| --- |
| определить концепт Numeric: тип T должен быть целым или с плавающей точкой  определить функцию print\_range для вектора типа Numeric:  для каждого элемента v в векторе, отфильтрованного по условию v > 0:  вывести v  вывести новую строку  создать вектор {-1, 2, 3, -4, 5}  вызвать print\_range для вектора |

**Пример кода 2 (C++20)**: Простая корутина для генерации чисел.

|  |
| --- |
| #include <iostream>  #include <coroutine>  struct Generator {  struct promise\_type {  int current\_value;  std::suspend\_always yield\_value(int value) {  current\_value = value;  return {};  }  std::suspend\_always initial\_suspend() { return {}; }  std::suspend\_always final\_suspend() noexcept { return {}; }  void unhandled\_exception() {}  Generator get\_return\_object() { return Generator{ this }; }  void return\_void() {}  };  using handle\_type = std::coroutine\_handle<promise\_type>;  handle\_type coro;  Generator(promise\_type\* p) : coro(handle\_type::from\_promise(\*p)) {}  ~Generator() { if (coro) coro.destroy(); }  int value() { return coro.promise().current\_value; }  bool next() { coro.resume(); return !coro.done(); }  };  Generator generate\_numbers(int n) {  for (int i = 0; i < n; ++i) {  co\_yield i;  }  }  int main() {  auto gen = generate\_numbers(5);  while (gen.next()) {  std::cout << gen.value() << " ";  }  std::cout << "\n";  } |

**Псевдокод 2**:

|  |
| --- |
| определить структуру Generator с корутиной  определить функцию generate\_numbers(n):  для i от 0 до n:  передать i как значение корутины  создать генератор для 5 чисел  пока генератор имеет следующее значение:  вывести текущее значение  вывести новую строку |

**Практическое применение**: C++20 активно используется в крупных проектах. Например, модули применяются в игровых движках для ускорения компиляции, корутины – в сетевых библиотеках, таких как Boost.Asio, для асинхронной обработки запросов, а ranges – в аналитических системах для обработки больших данных.

C++20 поднял язык на новый уровень, сделав его более модульным, выразительным и пригодным для асинхронного программирования.

# **6. C++23: Перспективы и доработки**

C++23, финализированный в 2023 году, продолжает развивать идеи C++20, добавляя удобные инструменты и готовя язык к будущим инновациям.

* **Улучшения модулей**: Упрощённая интеграция стандартной библиотеки в модульном виде, что повышает переносимость кода.
* **Расширения стандартной библиотеки**: Введение std::expected для обработки ошибок, std::flat\_map и std::flat\_set для оптимизированных словарей и множеств.
* **Форматирование текста**: Библиотека std::format предоставила мощный и гибкий инструмент для форматирования строк, заменив устаревший sprintf.
* **Рефлексия**: Экспериментальная поддержка рефлексии для метапрограммирования, позволяющая анализировать структуру кода на этапе компиляции.
* **Многопоточность**: Улучшения в std::jthread (автоматическое завершение потоков) и std::atomic для конкурентности.
* **Стековые корутины**: Упрощённая работа с корутинами для систем с ограниченными ресурсами.

**Пример кода 1 (C++23)**: Использование std::format и std::expected.

|  |
| --- |
| #include <iostream>  #include <format>  #include <expected>  std::expected<int, std::string> divide(int a, int b) {  if (b == 0) return std::unexpected("Division by zero");  return a / b;  }  int main() {  auto result = divide(10, 2);  if (result) {  std::cout << std::format("Result: {}\n", \*result);  }  else {  std::cout << std::format("Error: {}\n", result.error());  }  } |

**Псевдокод 1**:

|  |
| --- |
| определить функцию divide(a, b):  если b == 0, вернуть ошибку "Division by zero"  вернуть a / b  вызвать divide(10, 2)  если результат успешен:  вывести отформатированную строку с результатом  иначе:  вывести отформатированную строку с ошибкой |

**Пример кода 2 (C++23)**: Использование std::flat\_map.

|  |
| --- |
| #include <iostream>  #include <flat\_map>  int main() {  std::flat\_map<std::string, int> scores = { {"Alice", 90}, {"Bob", 85}, {"Charlie", 95} };  scores.insert({ "David", 88 });  for (const auto& [name, score] : scores) {  std::cout << std::format("{}: {}\n", name, score);  }  } |

**Псевдокод 2**:

|  |
| --- |
| создать flat\_map с парами {("Alice", 90), ("Bob", 85), ("Charlie", 95)}  добавить пару ("David", 88)  для каждой пары (name, score) в flat\_map:  вывести отформатированную строку с name и score |

**Практическое применение**: C++23 используется в проектах, требующих высокой производительности и читаемости. Например, std::format упрощает логирование в серверных приложениях, а std::expected применяется в обработке ошибок в сетевых протоколах. Рефлексия, хотя и экспериментальная, уже тестируется в инструментах для автоматической генерации кода.

C++23 делает язык более удобным и готовит его к будущим инновациям, таким как полноценная рефлексия и улучшенная конкурентность.

# **7. Сравнительный анализ и влияние на разработку**

Сравнение стандартов C++11/14/17/20/23 показывает их уникальный вклад в развитие языка:

* **Производительность**: C++11 внедрил многопоточность, C++17 добавил параллельные алгоритмы, C++20 оптимизировал компиляцию через модули, а C++23 улучшил работу с памятью через std::flat\_map.
* **Удобство**: Лямбда-выражения (C++11), структурированные привязки (C++17), std::format (C++23) и ranges (C++20) упростили написание кода.
* **Масштабируемость**: Концепты и модули (C++20) сделали язык пригодным для крупных проектов, а улучшения модулей в C++23 повысили переносимость.
* **Безопасность**: Умные указатели (C++11), std::optional и std::variant (C++17), std::expected (C++23) снизили вероятность ошибок.

**Влияние на разработку**:

* **Игровая индустрия**: Unreal Engine использует C++20 для модулей и ranges, что ускоряет компиляцию и упрощает обработку данных в рендеринге.
* **Сетевые приложения**: Boost.Asio интегрирует корутины C++20 для асинхронной обработки запросов, повышая масштабируемость серверов.
* **Научные вычисления**: Параллельные алгоритмы C++17 и constexpr функции C++14 применяются для оптимизации симуляций и генерации констант.
* **Кроссплатформенные инструменты**: std::filesystem (C++17) и std::format (C++23) упрощают разработку утилит, таких как CMake.

**Пример влияния**: В проекте по разработке игрового движка использование модулей C++20 сократило время компиляции на 30%, а корутины позволили реализовать асинхронную загрузку ресурсов, улучшив производительность. В финансовых системах std::expected (C++23) заменил традиционные коды ошибок, упростив обработку транзакций.

**Перспективы**: Ожидается, что C++26 внедрит полноценную рефлексию, что позволит создавать инструменты для автоматической генерации кода, и улучшит конкурентность через std::execution, упрощая параллельное программирование. Эти изменения укрепят позиции C++ в индустрии.

# **8. Заключение**

Современные стандарты C++11, C++14, C++17, C++20 и C++23 радикально преобразили язык, сделав его мощным, безопасным и универсальным инструментом. От умных указателей и лямбда-выражений в C++11 до модулей, корутин и концептов в C++20, каждый стандарт внёс вклад в эволюцию языка. C++23 продолжает этот путь, добавляя удобные инструменты, такие как std::format и std::expected, и готовя почву для будущих инноваций, таких как рефлексия. Эти изменения упростили разработку, расширили области применения C++ – от системного программирования до высокоуровневых приложений – и сохранили его конкурентоспособность на фоне современных языков. Перспективы развития C++ остаются многообещающими, и будущие стандарты продолжат укреплять его роль в индустрии программирования.