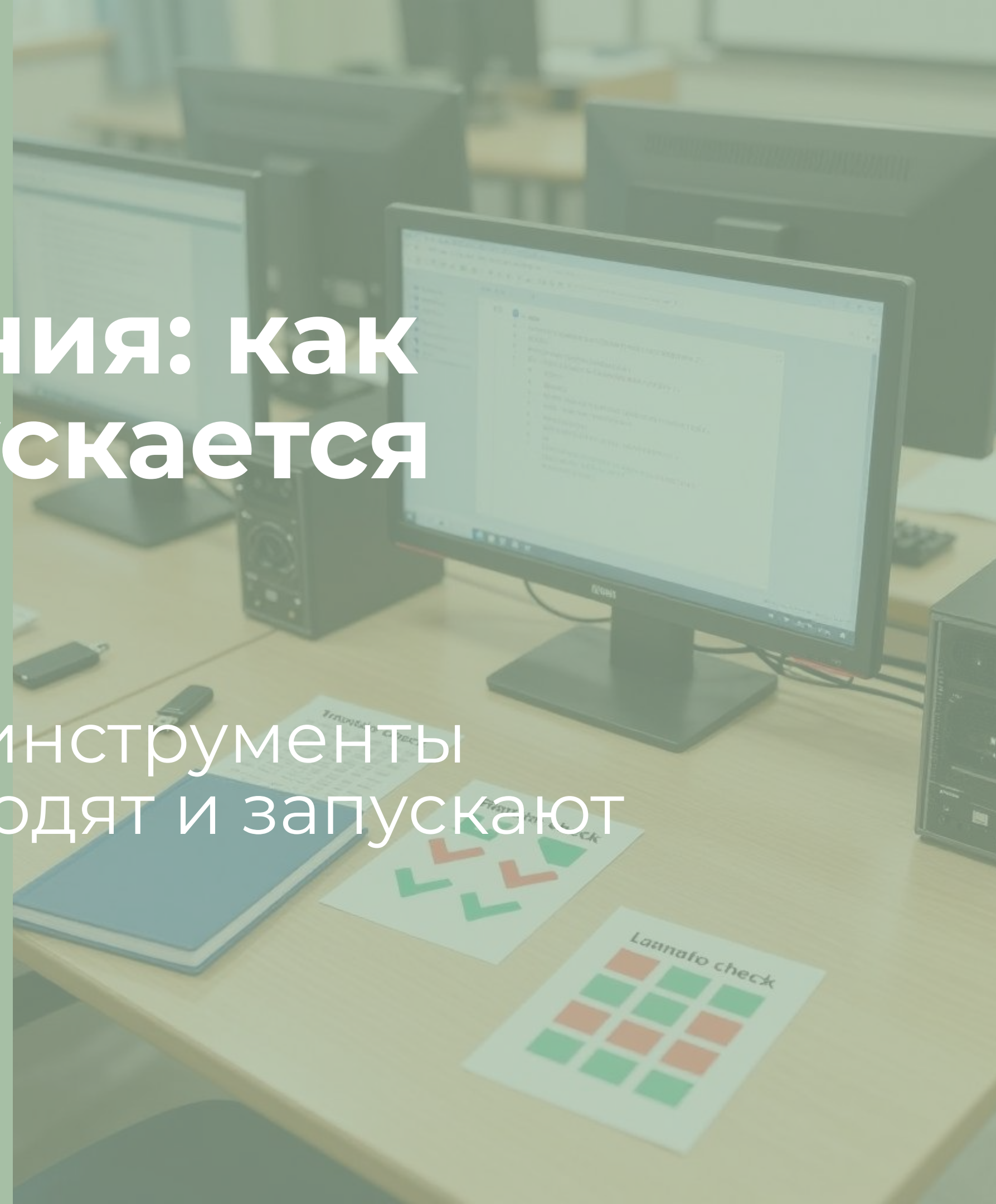


# Системы программирования: как создаётся и запускается программа

От кода до приложения: инструменты  
пишут, проверяют, переводят и запускают  
программу.





# Системы программирования: зачем они нужны в 10 классе

Системы программирования — фундамент любого программного продукта и школьных проектов. В 1950-х появились первые компиляторы, а сегодня IDE объединяет множество функций. На уроках информатики это помогает понять разницу между кодом, программой и средой исполнения.

# Основные компоненты системы программирования



## Текстовый редактор

Текстовый редактор нужен для ввода и правки исходного кода. Он помогает организовать программу по файлам и фрагментам, а также ускоряет работу за счёт подсказок по синтаксису и поиска.



## Транслятор

Транслятор переводит код на машинный или промежуточный язык. Это шаг, без которого невозможно перейти от записи на языке программиста к форме, понятной компьютеру.



## Компоновщик

Компоновщик собирает программу из полученных частей и связывает их между собой. Он обеспечивает корректное подключение необходимых модулей и определений, чтобы итоговый запуск был возможен.



## Загрузчик

Загрузчик подготавливает программу к запуску: размещает её в памяти и передаёт управление процессору. Благодаря ему выбранные фрагменты оказываются в нужном месте и порядке.



## Отладчик

Отладчик помогает находить ошибки в поведении программы. Он позволяет пошагово выполнять код, отслеживать значения переменных и проверять, как программа реагирует на разные входные данные.

# Путь от исходного текста к выполнению

Проверка и преобразование кода на последовательных стадиях — от синтаксиса до запуска на процессоре.

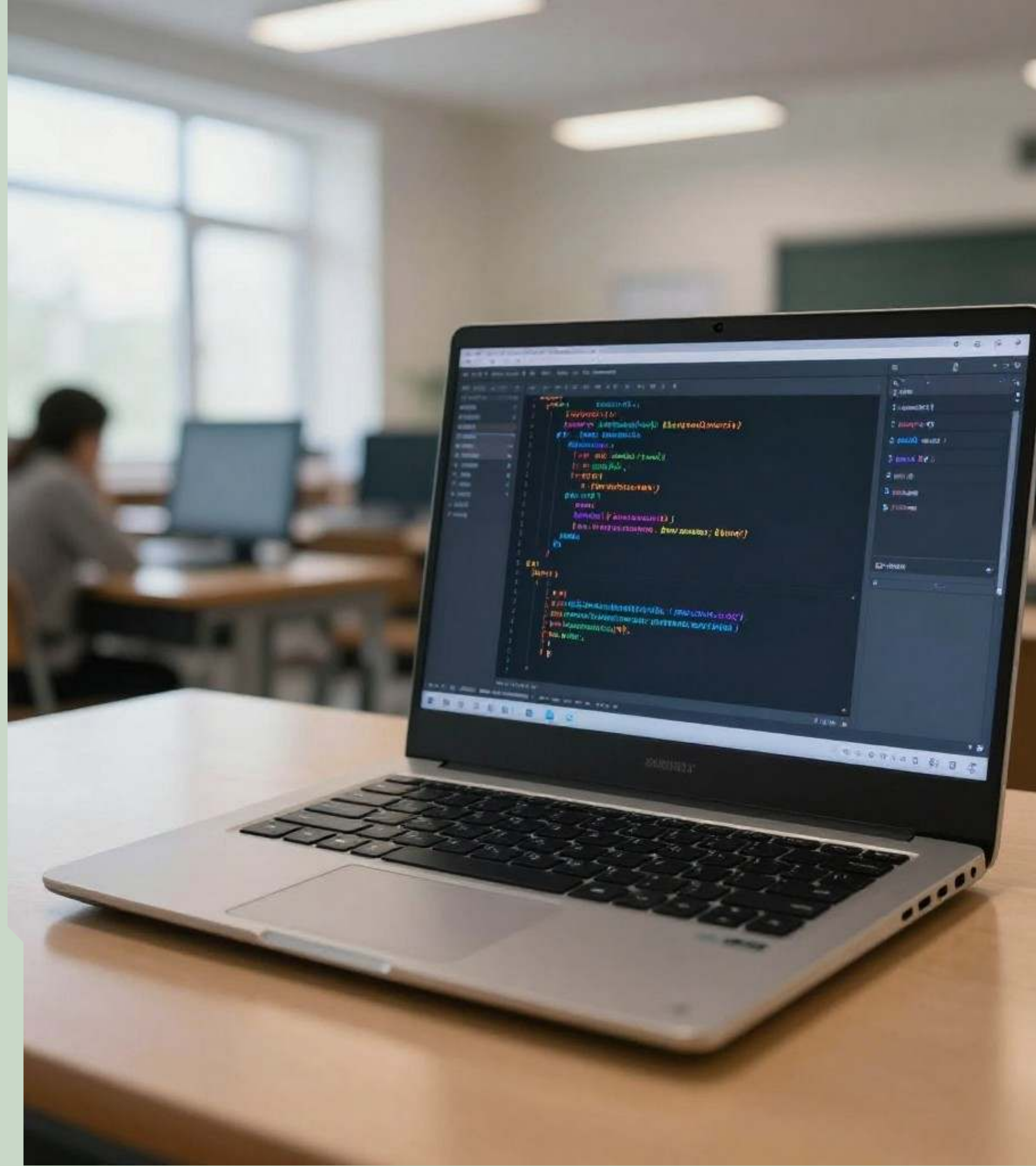
# Редактор кода и среда разработки

## Что делает редактор

Редактор кода поддерживает подсветку синтаксиса, нумерацию строк, автодополнение и быстрый поиск ошибок. В результате легче заметить опечатки и логические несостыковки ещё до запуска программы.

## Зачем нужна IDE

Интегрированная среда разработки (IDE) объединяет сборку проекта, запуск, отладку и работу с файлами в одном интерфейсе. Примеры: Visual Studio Code, PyCharm, IntelliJ IDEA и Eclipse.



# Сравнение компиляции, интерпретации и JIT-подхода

На практике разница ощущается не только по скорости, но и по тому, насколько быстро цикл «изменил — проверил» превращается в привычку.

Комбинация «перевод на лету + оптимизации» часто даёт компромисс: быстрее, чем чистая интерпретация, и гибче, чем полный предварительный перевод.



Обобщённое сравнение свойств подходов в учебных материалах и общих принципах трансляции (точные числа зависят от языка и реализации).

# Компилятор и интерпретатор: в чём разница

01

---

Компилятор переводит весь исходный код заранее и получает результат в виде отдельного исполняемого файла или объектного кода. Это позволяет запускать программу без повторного перевода на старте.

02

---

Интерпретатор выполняет программу по строкам или блокам, не формируя полноценный исполняемый файл на этапе запуска. Поэтому изменения можно проверять быстрее, особенно в учебных и экспериментальных сценариях.

03

---

При компиляции чаще выигрывает производительность выполнения, а при интерпретации — простота экспериментов. Выбор подхода зависит от требований к скорости старта, удобству правок и целевой платформы.



# Языки программирования и их типичные области применения

Ниже — примеры того, как язык связан со способом трансляции, задачами и характерной особенностью. Выбор всегда зависит от цели проекта и требований к производительности и срокам разработки.

Смысл выбора языка — не «лучше/хуже», а соответствие задаче: где важна скорость, где важна скорость разработки, и где нужна переносимость.

| Язык       | Тип трансляции                          | Основные области применения                  | Характерная особенность     |
|------------|-----------------------------------------|----------------------------------------------|-----------------------------|
| C          | компиляция                              | Системное ПО и драйверы                      | Контроль близ железу        |
| C++        | компиляция                              | Игры и высокопроизводительные приложения     | Опора на производительность |
| Java       | компиляция+среда выполнения             | Корпоративные системы и платформы обучения   | Переносимость через среду   |
| Python     | интерпретация (с возможной компиляцией) | Анализ данных, автоматизация, учебные задачи | Удобный синтаксис           |
| JavaScript | интерактивная веб-исполнение            | Интерактивные веб-страницы                   | Работа в браузере           |

Сводные учебные сведения по применению языков и их общим принципам трансляции (без привязки к конкретным версиям).

# Компоновщик, библиотеки и загрузчик

01

Компоновщик объединяет объектные файлы в единую программу и настраивает связи между модулями. Также он подключает библиотеки, чтобы нужные функции были доступны итоговому коду при запуске.

---

При статической компоновке компоненты включаются прямо в файл программы, а при динамической используются общие библиотеки операционной системы. Это влияет на размер файла и на то, как изменения библиотек сказываются на работе программы.

---

02

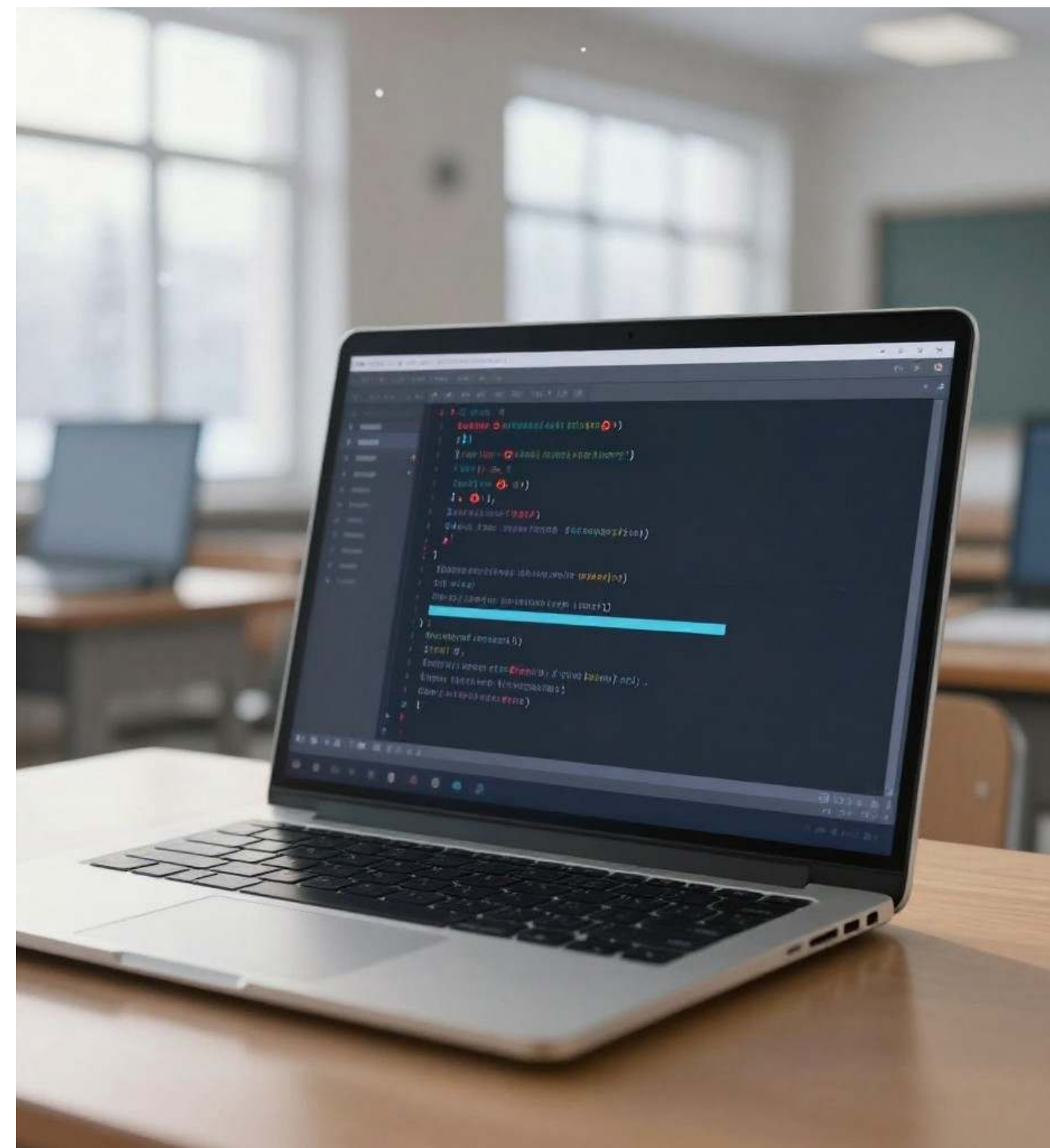
03

Загрузчик размещает программу в оперативной памяти и передаёт управление процессору. Так система начинает выполнение в нужной точке, имея корректно подготовленные данные и подгруженные зависимости.

---

# Отладка как способ поиска и исправления ошибок

- 01** Отладчик помогает уверенно разбираться с тем, что именно «ломается» в программе: можно ставить точки останова, выполнять код пошагово и наблюдать значения переменных. Такой подход снижает количество догадок при поиске причин.
- 02** Основные типы ошибок включают синтаксические, логические, ошибки времени выполнения и выход за границы массива. В школьных задачах отладка особенно полезна, когда нужно проверить циклы, условия и работу с массивами.



# Типичный цикл разработки программы

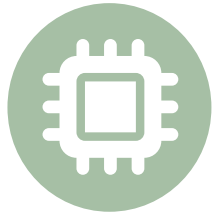
На практике этапы повторяются: результаты проверки и отладочных наблюдений подсказывают, что стоит уточнить алгоритм или переписать часть кода.

Диаграмма отражает цикличность разработки: чаще всего изменения требуются после тестирования и при уточнении решения на этапе проверки результата.



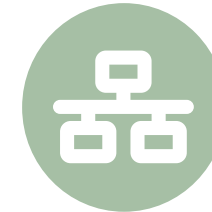
Обобщённая модель жизненного цикла разработки (учебная схема), без привязки к конкретным метрикам; актуальность на 2026 год.

# Среда выполнения и виртуальная машина



## Среда выполнения

Среда выполнения отвечает за запуск и корректную работу программы в конкретной операционной системе и на определённом устройстве. Она связывает приложение с доступными ресурсами и службами.



## Виртуальная машина

Виртуальная машина исполняет промежуточный код и тем самым отделяет программу от конкретной платформы. За счёт этого приложение легче переносить между разными компьютерами.



## Примеры технологий

Типичные варианты — Java Virtual Machine и .NET CLR. Они позволяют запускать программы на разных устройствах при наличии нужной среды выполнения и установленной платформы.

# Выбор системы программирования под задачу

Выбор делается по требованиям: скорости, типу интерфейса, характеру данных и масштабу проекта.

# Инструменты контроля качества кода

1

Линтеры автоматически проверяют оформление и типичные нарушения стиля: от пропусков точек с запятой до неверного именования переменных и несоблюдения принятых правил.

2

Статический анализ ищет потенциальные ошибки ещё до запуска: несогласованность типов, обращения к несуществующим значениям, подозрительные ветвления и возможные исключения.

3

Тестовые фреймворки помогают систематизировать проверку: запускать один и тот же набор тестов после изменений, чтобы подтверждать корректность функций и циклов.

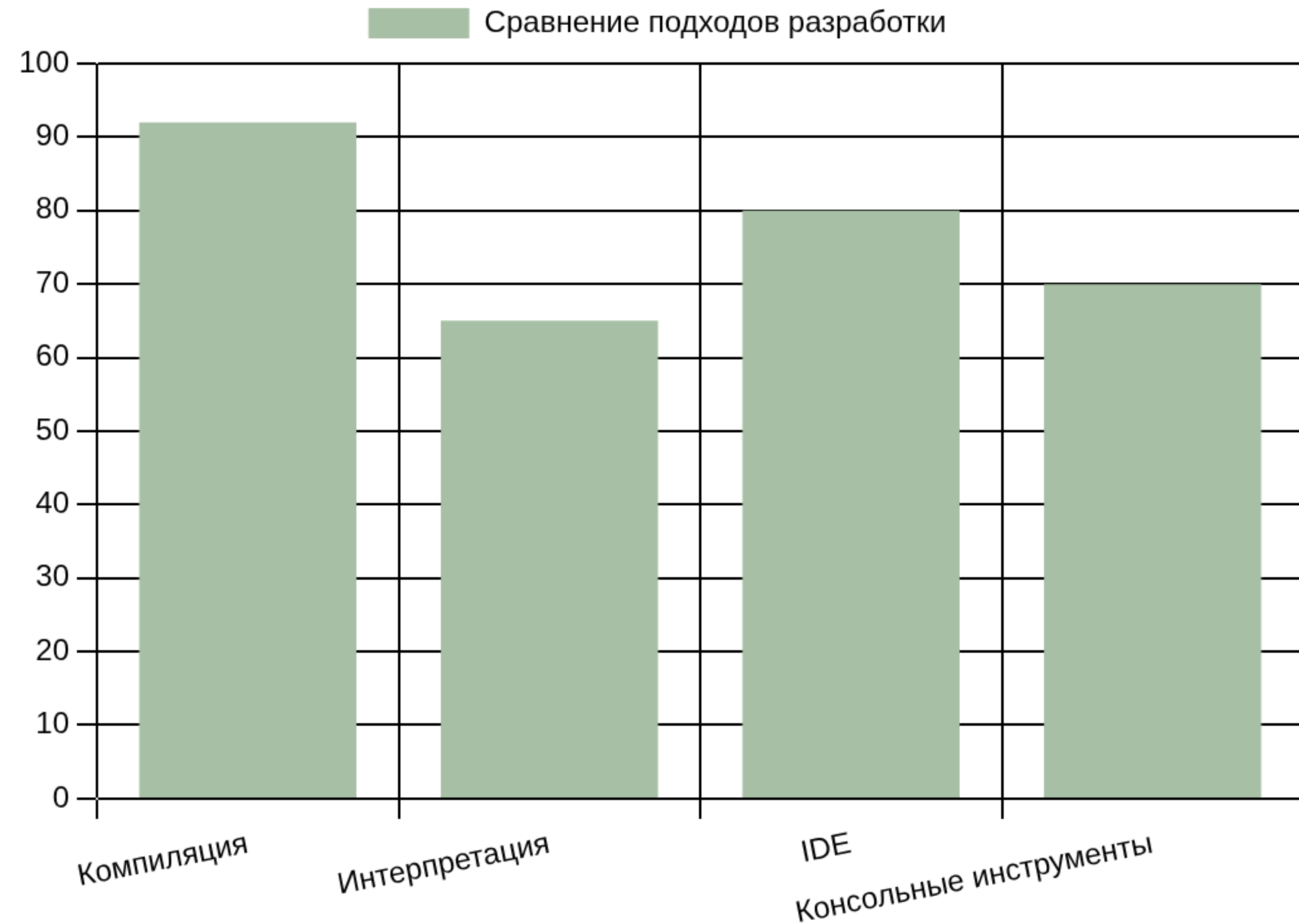
4

Профилировщики показывают, где программа тратит больше времени и памяти. Это помогает точнее оптимизировать «узкие места», а не заниматься случайными улучшениями.

## Преимущества и ограничения разных подходов

Выбор подхода обычно компромисс: если важна скорость работы, удобнее компиляция; если важнее быстрые пробы и обучение — интерпретация или консольные сценарии.

Сравнение показывает, что универсального варианта нет: продуктивность разработчика и требования к исполнению определяют наиболее подходящий инструмент.



*Качественная сравнительная шкала для учебного обсуждения (условные баллы); без привязки к конкретным продуктам и измерениям.*

# Системы программирования в реальных проектах

## 01 Игры, веб и отладка в процессе разработки

---

В разработке игр часто используют C++ и игровые движки со встроенными средствами отладки. Это помогает анализировать поведение объектов и исправлять ошибки в сложных сценах.

## 02 Научные задачи и встроенные системы

---

В научных и учебных проектах нередко применяют Python вместе с библиотеками NumPy и Pandas для работы с данными. Во встроенных системах чаще выбирают C, так как он теснее связан с аппаратным уровнем.



# Как системы программирования помогают изучать алгоритмы

## Запуск на учебной среде

Пошаговый запуск помогает наблюдать, как алгоритм «двигается» по строкам кода. Одновременно становятся видны изменения переменных и результат каждого вычисления.

## Анализ циклов

При просмотре работы циклов видно, когда тело цикла повторяется и почему завершается. Удобно отслеживать счётчики, условия выхода и накопление значений в массиве.

## Трассировка ветвлений

Трассировка условий позволяет проверить, какая ветка выполняется. Это особенно полезно при обучении логике if, обработке исключительных случаев и корректной проверке условий.

## Проверка рекурсии и массивов

Просмотр стека вызовов и значений аргументов делает рекурсию более понятной. Для массивов ясно видно обращения по индексам, что снижает число типичных логических ошибок.

# Ключевые компоненты и их функции

| Компонент   | Основная функция           | Что проверяется или обрабатывается                  |
|-------------|----------------------------|-----------------------------------------------------|
| Редактор    | Ввод и редактирование кода | Синтаксис подсказками, структура файла              |
| Компилятор  | Перевод программы          | Проверка формальных правил и генерация кода         |
| Компоновщик | Сборка модулей             | Связи между частями и подключение модулей           |
| Отладчик    | Поиск и исправление ошибок | Точки останова, значения переменных, ход выполнения |

Обобщённая схема состава системы программирования (учебный курс информатики).

Система программирования — это набор взаимосвязанных инструментов. Каждый компонент решает свою задачу и вместе они обеспечивают полный путь от текста к работающему результату.

Данные показывают, что качество результата обеспечивается на разных уровнях: от подготовки текста до проверки выполнения. Ошибки устраняются не одним действием, а последовательным контролем.

## Типичные проблемы при работе с программой и пути их устранения

Диагностика идёт по характеру ошибки: от синтаксиса и сборки до времени выполнения, производительности и переносимости.

# Превратите идею в результат: зачем нужны системы программирования

Системы программирования помогают превратить замысел в работающую программу: поддерживают создание, проверку, запуск и сопровождение кода. Они ускоряют поиск ошибок и повышают переносимость — это важная база для дальнейших тем информатики.