



Язык C++. Синтаксис

Презентация разработана в рамках гранта «Грант на обучение студентов по образовательным программам высшего образования для топ-специалистов в сфере информационных технологий. Договор № 70-2025-000850 с АНО «Аналитический центр при Правительстве Российской Федерации»

Александра Волосова,
к.т.н., доцент кафедры
ИУ5

План презентации

1. Базовые элементы синтаксиса C++
2. Основные правила синтаксиса
3. Распространенные синтаксические ошибки
4. Критерии эффективности синтаксиса
5. Стили программирования и лучшие практики
6. Современный C++ (C++11/14/17/20)

Что такое синтаксис?

Определение: Синтаксис - набор правил, определяющих структуру программ

Ключевые аспекты:

- Отвечает на вопрос: "Как писать корректный код?"
- Отличается от семантики ("Что делает код?")
- Включает лексемы, выражения, операторы, структуры управления

Синтаксис определяет грамматику языка программирования, обеспечивая однозначное понимание кода компилятором и разработчиками.

1. Базовые элементы синтаксиса

Основные компоненты C++ синтаксиса:

- Идентификаторы (имена переменных, функций, классов)
- Ключевые слова (if, for, class, void, etc.)
- Литералы (числа, строки, символы)
- Операторы (+, -, *, /, =, ==, etc.)
- Разделители (;, {}, (), [], ,)
- Комментарии (//, /* */)
- Директивы препроцессора (#include, #define)

Понимание базовых элементов - фундамент для написания корректного C++ кода.

Типы данных в C++

Классификация типов данных:

Встроенные типы:

- int, float, double, char, bool
- short, long, long long
- unsigned модификаторы

Пользовательские типы:

- class, struct, union
- enum, enum class
- typedef, using

Сложные типы:

- Массивы
- Указатели и ссылки
- Шаблоны

Система типов C++ обеспечивает безопасность и эффективность работы с данными.

Объявление переменных

// Различные способы объявления переменных

```
int x = 10; // традиционная инициализация
double y{3.14}; // uniform initialization (C++11)
auto z = "hello"; // auto type deduction
const int MAX_SIZE = 100; // константа
static int counter = 0; // статическая переменная
extern int global_var; // внешнее объявление
```

// Указатели и ссылки

```
int* ptr = &x;
int& ref = x;
```

Операторы C++

Классификация операторов:

- Арифметические: + - * / % ++ --
- Операторы сравнения: == != < > <= >=
- Логические: && || !
- Битовые: & | ^ ~ << >>
- Присваивания: = += -= *= /=
- Другие: sizeof, ternary operator ?:, scope resolution ::, member access . ->

Операторы позволяют манипулировать данными и управлять потоком выполнения программы

Структуры управления

// Условные операторы

```
if (x > 0) {  
    // do something  
} else if (x < 0) {  
    // do something else  
} else {  
    // default case}
```

// Циклы

```
for (int i = 0; i < 10; ++i) {  
    cout << i << endl;}
```

```
while (condition) {  
    // loop body}
```

```
do {  
    // loop body  
} while (condition);
```

Функции

// Объявление функции

```
int add(int a, int b);
```

// Определение функции

```
int add(int a, int b) {  
    return a + b;  
}
```

// Параметры по умолчанию

```
void print(string message, bool newline = true);
```

// Перегрузка функций

```
void print(int number);  
void print(double number);
```

// Шаблонная функция

```
template<typename T>  
T max(T a, T b) {  
    return a > b ? a : b;  
}
```

Классы и структуры

```
class MyClass {
private:
    int private_data;

protected:
    string protected_data;

public:
    // Конструктор
    MyClass(int data) : private_data(data) {}

    // Деструктор
    ~MyClass() {}

    // Метод
    void print() const {
        cout << private_data << endl;
    }

    // Статический метод
    static int get_count() {
        return count;
    }
};

// Наследование
class Derived : public MyClass {
    // наследует protected и public члены
};
```

Шаблоны (Templates)

```
// Шаблонная функция
template<typename T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

// Шаблонный класс
template<typename T, int Size>
class Array {
private:
    T data[Size];

public:
    T& operator[](int index) {
        return data[index];
    }
};

// Использование
Array<int, 10> int_array;
Array<double, 5> double_array;
```

Пространства имен

```
// Определение namespace
namespace mylibrary {
    class Calculator {
    public:
        static int add(int a, int b) {
            return a + b;
        }
    };

    const double PI = 3.14159;
}

// Использование
int result = mylibrary::Calculator::add(5, 3);

// using directive
using namespace mylibrary;
double circumference = 2 * PI * radius;

// using declaration
using std::cout;
using std::endl;
```

Директивы препроцессора

```
// Включение заголовочных файлов
#include <iostream>
#include "myheader.h"

// Макросы
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define PI 3.14159

// Условная компиляция
#ifdef DEBUG
    #define LOG(msg) cout << msg << endl
#else
    #define LOG(msg)
#endif

// pragma directives
#pragma once // защита от многократного включения
```

2. Основные правила синтаксиса

Критические правила C++ синтаксиса:

- Каждая инструкция заканчивается точкой с запятой (;)
- Блоки кода ограничиваются фигурными скобками {}
- Идентификаторы чувствительны к регистру
- Переменные должны быть объявлены перед использованием
- Функции должны быть объявлены или определены
- Правила видимости (scope)
- Правила преобразования типов
- Правила перегрузки операторов
- Правила наследования
- Правила шаблонов

Соблюдение синтаксических правил обязательно для успешной компиляции кода

3. Распространенные синтаксические ошибки

Частые ошибки новичков:

- Отсутствие ; в конце инструкции
- Непарные скобки {}, (), []
- Необъявленные переменные
- Несоответствие типов
- Ошибки в директивах #include
- Неправильное использование указателей
- Ошибки в шаблонах
- Неправильное наследование
- Ошибки в перегрузке операторов
- Проблемы с видимостью (public/private/protected)

Понимание типичных ошибок ускоряет процесс обучения и отладки

Примеры ошибок

```
// Ошибка: отсутствие ;  
int x = 5 // ошибка!
```

```
// Ошибка: непарные скобки  
if (x > 0 {  
    // ошибка!  
}
```

```
// Ошибка: необъявленная переменная  
y = 10; // что такое y?
```

```
// Ошибка: несоответствие типов  
int* ptr = "hello"; // ошибка!
```

```
// Ошибка: неправильное наследование  
class Derived : private Base {};  
Derived d;  
Base* b = &d; // ошибка!
```

4. Эффективность синтаксиса

Критерии эффективности синтаксиса:

1. Читаемость кода:

- Понятные имена переменных
- Правильное форматирование
- Логическая структура

2. Производительность:

- Оптимальное использование ресурсов
- Минимизация накладных расходов

3. Поддерживаемость:

- Легкость внесения изменений
- Минимизация ошибок

4. Расширяемость:

- Возможность добавления нового функционала

Эффективный синтаксис балансирует между выразительностью и производительностью

Читаемость кода

// ПЛОХО: непонятные имена, плохое форматирование

```
int f(int a,int b){return a>b?a:b;}
```

// ХОРОШО: понятные имена, хорошее форматирование

```
int find_maximum(int first_number, int second_number) {  
    if (first_number > second_number) {  
        return first_number;  
    } else {  
        return second_number;  
    }  
}
```

// ЛУЧШЕ: используем стандартную функцию

```
int maximum = std::max(first_number, second_number);
```

5. Стили программирования

1. C++ Core Guidelines Style

Стиль, продвигаемый Бьярном Страуструпом и комитетом по стандартизации C++

```
// Имена типов - PascalCase
class NetworkConnection {
public:
    // Функции - camelCase
    void establishConnection();
    // Константы - UPPER_CASE
    static constexpr int MAX_RETRIES = 5;
private:
    // Члены класса - suffix_
    std::string name_;
    int timeoutMs_;
};
// Параметры шаблонов - T, U или с префиксом T
template<typename TValue, typename TKey>
class HashMap {
    // ...
};
```

C++ Core Guidelines

Рекомендации от создателей C++:

- P.1: Выражайте идеи непосредственно в коде
- P.2: Пишите на стандартном C++
- P.3: Выражайте intent
- I.2: Избегайте неконстантных глобальных переменных
- F.15: Предпочитайте простые и обычные способы
- C.1: Организуйте связанные данные в структуры
- ES.20: Всегда инициализируйте объекты
- ES.23: Предпочитайте {} инициализацию
- R.1: Управляйте ресурсами автоматически
- SL.1: Используйте библиотеки, когда возможно

Теория: Core Guidelines представляют современные best practices для C++.

Р — Philosophy (Философия)

Р.1: Выражайте идеи непосредственно в коде

// ПЛОХО: Намерение скрыто

```
if (x > 0 && y > 0 && z > 0) { ... }
```

// ХОРОШО: Намерение явное

```
bool all_positive = x > 0 && y > 0 && z > 0; if (all_positive) { ... }
```

// Еще лучше - вынести в функцию

```
bool are_all_positive(int a, int b, int c)  
{ return a > 0 && b > 0 && c > 0;}
```

Р.2: Пишите на стандартном C++

// ПЛОХО: Нестандартные расширения

```
#ifdef _MSC_VER
```

```
__declspec(dllexport) void func();
```

```
#endif
```

// ХОРОШО: Стандартный C++

```
export void func(); // C++20 модули
```

Р.3: Выражайте intent (намерение)

// ПЛОХО: Неясное намерение

```
void process(int* data, int n) { ... }
```

// ХОРОШО: Ясное намерение

```
void process_sensor_readings(std::span<int> readings) { ... }
```

I — Interfaces (Интерфейсы)

I.2: Избегайте неконстантных глобальных переменных

// ПЛОХО: Глобальное состояние

```
int global_counter = 0;
```

```
void increment() {
```

```
    global_counter++;
```

// Потоконебезопасно, сложно тестировать

```
}
```

// ХОРОШО: Локальное состояние или dependency injection

```
class Processor
```

```
{ int counter_ = 0;
```

```
    public: void increment()
```

```
        { counter_++; }
```

```
};
```

F — Functions (Функции)

F.15: Предпочитайте простые и обычные способы

// ПЛОХО: Слишком сложно

```
auto result = std::accumulate(v.begin(), v.end(), 0,  
    [](int a, int b) { return a + b; });
```

// ? ХОРОШО: Просто и понятно

```
int sum = 0; for (int x : v) {  
    sum += x; }
```

// Или еще лучше в C++23

```
int sum = std::ranges::fold_left(v, 0, std::plus{});
```

C — Classes and class hierarchies (Классы)

C.1: Организуйте связанные данные в структуры

// ПЛОХО: Разрозненные данные

```
std::string user_name;  
int user_age;  
std::string user_email;
```

// ХОРОШО: Связанные данные вместе

```
struct User {  
    std::string name;  
    int age;  
    std::string email;};  
User current_user{"John", 25, "john@example.com"};
```

ES — Expressions and statements (Выражения)

ES.20: Всегда инициализируйте объекты

// ПЛОХО: Неинициализированная переменная

```
int x; // Мусорное значение  
use(x); Неопределенное поведение
```

// ХОРОШО: Всегда инициализировать

```
int x = 0;  
int y{}; // Инициализация по умолчанию  
int z{42}; // Прямая инициализация
```

ES.23: Предпочитайте {} инициализацию

// ПЛОХО: Старая инициализация

// Прямая инициализация

double d = 3.14;

int x(d); *// ОПАСНО: сужающее преобразование. Компилятор может предупредить, но разрешит*
// x будет равно 3 (потеря дробной части)

// Копирующая инициализация

double d = 3.14;

int y = d; *// ТА ЖЕ ПРОБЛЕМА: сужающее преобразование, y будет равно 3*

// ХОРОШО: Uniform initialization

double d = 3.14;

int x{d}; *// КОМПИЛЯТОР ЗАПРЕТИТ: ошибка компиляции!*
// error: narrowing conversion of 'd' from 'double' to 'int'

//ПЛОХО: Старый способ (может быть неочевидным)

std::vector<int> v1(5, 10); *// 5 элементов со значением 10: {10, 10, 10, 10, 10}*

// ХОРОШО: Современный способ (явный и понятный)

std::vector<int> v2{5, 10}; *// 2 элемента: 5 и 10: {5, 10}*

R — Resource management (Управление ресурсами)

R.1: Управляйте ресурсами автоматически

// ПЛОХО: Ручное управление памятью

```
int* data = new int[100];
```

// ... использование ...

```
delete[] data; // Легко забыть!
```

// ХОРОШО: Автоматическое управление

```
std::vector<int> data(100);
```

// Память освободится автоматически

```
std::unique_ptr<int[]> smart_data = std::make_unique<int[]>(100);
```

SL — Support library (Стандартная библиотека)

SL.1: Используйте библиотеки, когда возможно

// ПЛОХО: изобретение велосипеда — написание собственной реализации там, где уже есть проверенные и оптимизированные решения в стандартной библиотеке

```
void my_sort(std::vector<int>& v) {
```

```
// Собственная реализация сортировки}
```

// ХОРОШО: Использование стандартной библиотеки

```
std::sort(v.begin(), v.end());
```

```
std::ranges::sort(v);
```

// C++20 - еще лучше!

// Использование std::array вместо сырых массивов

```
std::array<int, 100> data; // вместо int data[100];
```

2. Google C++ Style

Строгий стиль, используемый в Google

// Имена классов - CamelCase

```
class MyClass {
```

```
public: // Отступ 1 пробел
```

// Функции - CamelCase

```
void DoSomething();
```

// Переменные - lowercase_with_underscores

```
int member_variable;
```

// Константы - kCamelCase

```
static constexpr int kMaxConnections = 10;
```

```
private:
```

// Приватные члены - suffix_

```
std::string name_; };
```

// Макросы - UPPER_CASE (но не рекомендуется)

```
#define ROUND(x) ((x) + 0.5)
```

Google C++ Style Guide

Основные рекомендации Google:

- Отступы: 2 пробела
- Длина строки: 80 символов
- Имена переменных: snake_case
- Имена классов: CamelCase
- Константы: kConstantName
- Фигурные скобки: на той же строке
- Использование using запрещено в .h файлах
- Исключения: не используются
- RTTI: не используется
- Умные указатели: предпочтительны

Style Guide обеспечивает единообразие кода в больших проектах и командах

3. LLVM/Clang Style

Стиль, используемый в проектах LLVM

```
// Имена классов - CamelCase
class DataProcessor {
public:
    // Функции - camelCase
    void processData();

    // Переменные - camelCase
    int connectionTimeout;

    // Приватные члены - m_camelCase
private:
    std::string m_userName;
    int m_retryCount;
};

// Макросы - UPPER_CASE
#define DEBUG_MODE 1
```

4. Microsoft C++ Style

Стиль, распространенный в Windows-разработке

// Венгерская нотация (устаревшая, но встречается)

```
class CMyClass {  
public:  
    void DoSomething();  
};
```

// Префиксы: m_ - member, p - pointer, n - number

```
CString m_strName;  
int* m_pBuffer;  
int m_nCount;
```

// Интерфейсы с префиксом I

```
class IMyInterface {  
    virtual HRESULT QueryInterface() = 0;  
};
```

5. Modern C++ Enterprise Style

Современный стиль для крупных проектов

```
// Пространства имен для организации кода
namespace company::project::module {
    // Интерфейсы с суффиксом Interface
    class DatabaseInterface {
    public:
        virtual ~DatabaseInterface() = default;
        virtual Result connect() = 0;
    };
    // Реализации с суффиксом Impl
    class DatabaseImpl : public DatabaseInterface {
    public:
        Result connect() override;
    private:
        // Умные указатели вместо сырых
        std::unique_ptr<Connection> connection_;
        std::shared_ptr<Logger> logger_;
    };
} // namespace company::project::module
```

6. Functional C++ Style

Стиль с акцентом на функциональное программирование

cpp

// Много лямбд и функциональных конструкций

```
auto process_data = [](auto&& data) {  
    return data | std::views::transform([](int x) { return x * x; })  
        | std::views::filter([](int x) { return x > 0; })  
        | std::ranges::to<std::vector>();  
};
```

// constexpr везде где возможно

```
constexpr auto calculate = [](int a, int b) {  
    return a + b;  
};
```

7. Game Development Style

Стиль, распространенный в игровой индустрии

// Префиксы для быстрой идентификации

```
class FCharacter { // F - классы игровых объектов
```

```
    public:
```

```
        void Tick(float DeltaTime);
```

```
    private:
```

```
        FVector Location; // F - структуры данных
```

```
        FQuat Rotation;
```

```
        UMeshComponent* Mesh; // U - компоненты Unreal };
```

// Максимальная производительность

```
FORCEINLINE void Process() {
```

```
    // inline везде где возможно
```

```
}
```

Snake_case (змеиный_стиль)

//Все буквы строчные, слова разделяются подчеркиваниями

```
int user_age = 25;  
std::string first_name = "John";  
void calculate_total_price();  
const int MAX_RETRY_COUNT = 5;  
class database_connection;
```

Характеристики:

Все буквы в нижнем регистре

Слова разделяются подчеркиванием _

Читается как "user underscore age"

CamelCase (верблюжийСтиль)

// Слова пишутся слитно, каждое с заглавной буквы

```
int userAge = 25;  
std::string firstName = "John";  
void calculateTotalPrice();  
const int maxRetryCount = 5;  
Class DatabaseConnection;
```

Разновидности camelCase:

lowerCamelCase (для переменных и функций)

```
int userAge;           // первое слово с маленькой буквы  
void calculateTotal(); // первое слово с маленькой буквы
```

UpperCamelCase/PascalCase (для классов и типов)

```
class UserProfile;    // все слова с заглавной буквы  
struct DataPoint;     // все слова с заглавной буквы  
interface Database;   // все слова с заглавной буквы
```

CamelCase (верблюжийСтиль)

// Слова пишутся слитно, каждое с заглавной буквы

```
int userAge = 25;  
std::string firstName = "John";  
void calculateTotalPrice();  
const int maxRetryCount = 5;  
Class DatabaseConnection;
```

Разновидности camelCase:

lowerCamelCase (для переменных и функций)

```
int userAge;           // первое слово с маленькой буквы  
void calculateTotal(); // первое слово с маленькой буквы
```

UpperCamelCase/PascalCase (для классов и типов)

```
class UserProfile;    // все слова с заглавной буквы  
struct DataPoint;     // все слова с заглавной буквы  
interface Database;   // все слова с заглавной буквы
```

Тип	Snake_case	CamelCase
Переменная	user_age	userAge
Функция	calculate_total()	calculateTotal()
Класс	database_handler	DatabaseHandler
Константа	MAX_SIZE	maxSize или MaxSize
Параметр	input_data	inputData

RAII (Resource Acquisition Is Initialization)

Идиома: Ресурс выделяется в конструкторе и освобождается в деструкторе. Это гарантирует, что ресурсы всегда будут правильно освобождены

Без RAI

```
void process_file() {  
    * = fopen("data.txt", "r");
```

// Получение ресурса

```
if (! ) return;
```

// Работа с файлом...

```
if ( ) return;
```

// □ Утечка файлового дескриптора!

```
fclose( ); // Легко забыть!
```

```
}
```

C RAI

```
class FileRAII {  
    FILE* file_;  
public:  
    FileRAII(const char* filename, const char* mode)  
        : file_{fopen(filename, mode)} {}  
  
    ~FileRAII() {  
        if (file_) fclose(file_); // Автоматическое освобождение  
    }  
  
    // Запрещаем копирование  
    FileRAII(const FileRAII&) = delete;  
    FileRAII& operator=(const FileRAII&) = delete;  
  
    operator FILE*() const { return file_; }  
};  
  
void process_file() {  
    FileRAII file("data.txt", "r"); // Ресурс получен  
    if (!file) return;  
  
    // Работа с файлом...  
    if (error_occurred) return; // Деструктор автоматически закроет  
    файл!  
    // Не нужно вызывать fclose() - деструктор сделает это  
}
```

Ресурсы управляемые через RAII

1. Память (самый частый случай)

// ☒ Без RAII

```
void unsafe_memory() { int* data = new int[100];
```

```
// ... использование ... delete[] data;
```

```
// Легко забыть! }
```

// С RAII

```
void safe_memory()
```

```
{ std::vector<int> data(100);
```

```
// RAII!
```

```
std::unique_ptr<int[]> smart_data = std::make_unique<int[]>(100);
```

```
// Память освободится автоматически }
```

2. Файлы

```
void safe_file_operations() {
```

```
    std::ifstream file("data.txt"); // RAII - закроется автоматически
```

```
    std::ofstream output("result.txt"); // То же самое // C++20: еще лучше
```

```
    std::jthread worker{[](){ /* поток завершится автоматически */ }}; }
```

3. Сетевые соединения

```
class DatabaseConnection {
```

```
    Connection* conn_;
```

```
public:
```

```
    DatabaseConnection() : conn_{connect_to_db()} {}
```

```
    ~DatabaseConnection() { disconnect(conn_); } // Автоматическое отключение // ... методы работы с соединением ... };
```

```
    void query_database() { DatabaseConnection db; // Подключение установлено
```

```
        // Выполняем запросы... // При выходе из функции соединение автоматически закроется }
```

4. Мьютексы и блокировки

```
std::mutex mtx;
```

```
void thread_safe_function() {
```

```
    // Опасный подход
```

```
    mtx.lock(); // ... критическая секция ...
```

```
    mtx.unlock(); // Легко забыть!
```

```
    // RAII подход
```

```
    std::lock_guard<std::mutex> lock(mtx); // Блокировка в конструкторе // ... критическая секция ... // Авторазблокировка в деструкторе }
```

Преимущества RAII

- ❑ **Исключительная безопасность** — ресурсы освобождаются даже при исключениях
- ❑ **Отсутствие утечек** — деструкторы вызываются автоматически
- ❑ **Чистый код** — не нужно помнить о ручной очистке
- ❑ **Безопасность потоков** — правильное управление блокировками

Const Correctness

Принцип проектирования в C++, который означает правильное и последовательное использование ключевого слова `const` для обеспечения безопасности типов и предотвращения непреднамеренных изменений данных

1. Const переменные

```
const int MAX_SIZE = 100;           // Неизменяемая переменная
const double PI = 3.14159;          // Значение не может быть изменено
// Ошибка компиляции
MAX_SIZE = 200; // Cannot assign to variable 'MAX_SIZE' with const-qualified type
```

2. Указатели и const

```
int value = 10; // Указатель на константу
const int* ptr1 = &value; // Данные нельзя менять через ptr1 // *ptr1 = 20;
Ошибка // Константный указатель
int* const ptr2 = &value; // Указатель нельзя перенаправить
// ptr2 = nullptr;
Ошибка // Константный указатель на константу
const int* const ptr3 = &value; // Ничего нельзя менять
```

3. Const ссылки

```
срр
int x = 5;
const int& ref = x; // Ссылка на константу
// ref = 10;
Ошибка - нельзя менять данные через ref
x = 10; // Можно - оригинальная переменная не const
```

Const в функциях

4. Параметры функций

// Правильно: передаем по const ссылке (эффективно и безопасно)

```
void print_vector(const std::vector<int>& vec) {  
    // vec.push_back(1); Ошибка - нельзя изменять  
    for (int num : vec) {  
        std::cout << num << " ";  
    }  
}
```

// Плохо: неконстантная ссылка без необходимости

```
void modify_vector(std::vector<int>& vec) {  
    vec.push_back(1); // Изменяет оригинальный вектор}
```

5. Const в шаблонах

```
template<typename T>
void process(const T& container) {
    // Гарантируем, что не изменим контейнер
    for (const auto& item : container) {
        // Обработка каждого элемента
    }
    // constexpr + const (C++17+)
    constexpr int compile_time_computation() {
        const int x = 10;
        const int y = 20;
        return x + y;
    }
```



Преимущества Const Correctness

- ❑ **Безопасность** — предотвращает случайные изменения
- ❑ **Ясность кода** — явно показывает намерения
- ❑ **Оптимизация** — компилятор может лучше оптимизировать
- ❑ **Многопоточность** — const объекты потокобезопасны для чтения
- ❑ **Интерфейс дизайн** — четко разделяет методы на изменяющие и неизменяющие

6. Современный C++ (C++11/14/17/20)

Ключевые особенности современных стандартов:

1. auto - автоматическое выведение типа
2. range-based for loops
3. lambda-выражения
4. умные указатели (smart pointers (unique_ptr, shared_ptr))
5. семантика перемещения (move semantics)
6. constexpr
7. nullptr
8. enum класс (class)
9. static_assert
10. variadic шаблоны (templates)
11. concepts (C++20)
12. ranges (C++20)
13. coroutines (C++20)
14. modules (C++20)

Современный C++ предлагает более выразительный и безопасный синтаксис.

1. auto — автоматическое выведение типа

auto позволяет компилятору автоматически определять тип переменной на основе её значения. Это упрощает код и делает его более читаемым.

Пример:

```
auto x = 5; // x имеет тип int
auto y = "hello"; // y имеет тип const char*
```

2. Range-based for loops

Циклы на основе диапазонов упрощают перебор элементов контейнеров.

Пример:

```
std::vector<int> v = {1, 2, 3};
for (auto& x : v) { x *= 2; // Модификация элементов }
```

3. Lambda-выражения

Лямбда-выражения позволяют создавать анонимные функции прямо в коде.

Пример:

```
auto sum = [](int a, int b) { return a + b; };
std::cout << sum(2, 3); // Выведет 5
```

4. Умные указатели (unique_ptr, shared_ptr)

Умные указатели управляют памятью автоматически, предотвращая утечки.

Пример:

```
std::unique_ptr<int> ptr(new int(10)); // Исключительное владение
std::shared_ptr<int> shared(new int(20)); // Совместное владение
```

5. Семантика перемещения (move semantics)

Позволяет эффективно передавать ресурсы между объектами, минимизируя копирование.

Пример:

```
std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2 = std::move(v1); // v1 становится пустым
```

6. constexpr

Позволяет вычислять значения на этапе компиляции.

Пример:

```
constexpr int factorial(int n) { return n <= 1 ? 1 : n * factorial(n - 1); }
constexpr int result = factorial(5); // result = 120
```

7. nullptr

Безопасный способ представления нулевого указателя.

Пример:

```
int* ptr = nullptr; // Вместо NULL или 0
```

8. Enum class

Scored перечисления предотвращают конфликты имён и обеспечивают безопасность типов.

Пример:

```
enum class Color { Red, Green, Blue }; Color c = Color::Red;
```

9. Static_assert

Проверяет условия на этапе компиляции.

Пример:

```
static_assert(sizeof(int) == 4, "Размер int должен быть 4 байта");
```

10. Variadic шаблоны

Шаблоны с переменным числом аргументов.

Пример:

```
template<typename... Args> void print(Args&&... args) { (std::cout << ... << args) << '\n';  
} print("Hello", " ", "World"); // Выведет "Hello World"
```

11. Concepts (C++20)

Ограничения на параметры шаблонов для улучшения читаемости и безопасности.

Пример:

```
template<typename T> concept Integral = std::is_integral_v<T>; template<Integral T> T  
add(T a, T b) { return a + b; }
```

12. Ranges (C++20)

Упрощают работу с последовательностями данных.

Пример:

```
#include <ranges> std::vector<int> v = {1, 2, 3, 4, 5}; auto even = v |  
std::views::filter([](int x) { return x % 2 == 0; }); for (int x : even) { std::cout << x << ' '; //  
Выведет 2 4 }
```

13. Coroutines (C++20)

Позволяют писать асинхронный код в синхронном стиле.

Пример:

```
#include <coroutine> task<int> async_add(int a, int b) { co_return a + b; } int  
main() { auto result = co_await async_add(2, 3); std::cout << result; //
```

Выведет 5 }

14. Modules (C++20)

Модули — это новая система организации кода, которая заменяет традиционную систему заголовочных файлов. Они позволяют:

Ускорить процесс компиляции

Избежать проблем с include-зависимостями

Улучшить организацию кода

Пример использования модулей:

```
// module.cppm export module MyModule; export int add(int a, int b) { return a + b; } //
```

```
main.cpp import MyModule; int main() { int result = add(2, 3); return 0; }
```

Преимущества модулей:

- **Производительность:** Компиляция происходит быстрее, так как модули компилируются один раз
- **Организация:** Более чёткое разделение интерфейса и реализации
- **Безопасность:** Меньше проблем с конфликтами имён

Важные особенности современного C++

Современный C++ предоставляет более выразительный и безопасный синтаксис благодаря:

Типобезопасности: nullptr, enum class, умные указатели

Метапрограммированию: constexpr, concepts, variadic templates

Функциональному стилю: lambda-выражения, ranges

Асинхронности: coroutines

Модульности: module system

Эти возможности позволяют писать более надёжный, производительный и поддерживаемый код, сохраняя при этом высокую эффективность и контроль над ресурсами

Лучшие практики (Best Practices)

Рекомендации по написанию качественного кода:

1. Всегда инициализируйте переменные
2. Используйте `const` где возможно
3. Предпочитайте `{}` инициализацию
4. Избегайте голых указателей
5. Используйте умные указатели
6. Следуйте правилу 3/5/0*
7. Используйте исключения для обработки ошибок
8. Пишите самодокументирующийся код
9. Тестируйте ваш код
10. Следуйте стандартам кодирования



Правило 3/5/0:

- **Правило трёх (Rule of Three)**
Если в классе требуется определить деструктор, конструктор копирования или оператор присваивания, то, скорее всего, потребуется определить все три. Это связано с тем, что компилятор автоматически генерирует эти функции, если они не определены пользователем. Однако, если класс управляет ресурсами (например, динамической памятью), стандартные версии этих функций могут работать некорректно.
- **Правило пяти (Rule of Five)**
В C++11 добавлены конструктор перемещения и оператор перемещения. Если класс требует одну из пяти специальных функций-членов (деструктор, конструктор копирования, оператор присваивания, конструктор перемещения, оператор перемещения), то, вероятно, потребуется определить все пять. Это правило помогает избежать утечек памяти и неэффективного копирования.
- **Правило нуля (Rule of Zero)**
Рекомендуется избегать определения всех пяти специальных функций-членов, если это возможно. Вместо этого следует использовать стандартные контейнеры и умные указатели (например, `std::unique_ptr` и `std::shared_ptr`), которые управляют ресурсами автоматически. Это упрощает код и делает его более безопасным.
- Пример:

```
class ResourceManager {  
public:  
    ResourceManager() { /* выделение ресурсов */ }  
    ~ResourceManager() { /* освобождение ресурсов */ }  
  
    // Следуем правилу нуля: не определяем остальные функции  
};
```

Ресурсы для изучения

Книги:

- "Язык программирования C++" - Бьярн Страуструп
- "Эффективный современный C++" - Скотт Мейерс
- "Учебник по C++" - Стэнли Липпман

Онлайн ресурсы:

- cppreference.com - полная информация
- learncpp.com - учебник для начинающих
- isocpp.org - официальный сайт C++

Статьи и руководства:

- **C++ Core Guidelines**
- Документация Microsoft C++