



Переменные и операторы в C++

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 лекции

1. Основные типы данных
2. Объявление и инициализация переменных
3. Константы и модификаторы
4. Арифметические операторы
5. Операторы присваивания
6. Операторы сравнения
7. Логические операторы
8. Побитовые операторы
9. Приоритет операторов
10. Преобразование типов
11. Ключевое слово `auto`
12. Best practices MIT

1. Основные типы данных C++

Целочисленные:

- `int`: -2^{31} до $2^{31}-1$ (4 байта)
- `short`: -32768 до 32767 (2 байта)
- `long`: 8 байт (64-bit)
- `long long`: 8 байт

Вещественные:

- `float`: 7 значащих цифр (4 байта)
- `double`: 15 значащих цифр (8 байт)
- `long double`: 19+ значащих цифр

Символьные:

- `char`: 1 байт (-128 до 127)
- `unsigned char`: 0 до 255

Логический:

- `bool`: `true` или `false`

2. Объявление переменных

```
// Синтаксис объявления:
```

```
// type variable_name;
```

```
int age;
```

```
// Объявление
```

```
float salary = 2500.50;
```

```
// Объявление с инициализацией
```

```
double pi{3.14159};
```

```
// Uniform initialization (C++11)
```

```
char grade = 'A';
```

```
bool is_valid = true;
```

```
// Множественное объявление:
```

```
int x, y, z;
```

```
// Не рекомендуется
```

```
int width = 10, height = 20;
```

```
// Правило MIT: одна переменная на строку
```

```
int student_count;
```

```
int course_credits;
```

```
int total_score;
```

3. Способы инициализации

```
// 1. Копирующая инициализация (традиционная)
```

```
int a = 5;
```

```
double d = 3.14;
```

```
// 2. Прямая инициализация
```

```
int b(10);
```

```
float f(2.71f);
```

```
// 3. Uniform initialization (C++11) - РЕКОМЕНДУЕТСЯ
```

```
int c{15}; // Не допускает narrowing
```

```
int e{}; // Value initialization (0)
```

```
double pi{3.14159};
```

```
// 4. Стандартная инициализация (C++20)
```

```
int g = {20}; // Аналогично uniform
```

```
// Преимущества {}:
```

- Запрещает сужающие преобразования
- Единообразный синтаксис
- Явная инициализация по умолчанию

4. Константы и модификаторы

```
// const: значение не может быть изменено
const int MAX_STUDENTS = 100;
const double PI = 3.14159;
```

```
// constexpr: вычисляется на этапе компиляции (C++11)
constexpr int ARRAY_SIZE = 100;
constexpr double GRAVITY = 9.81;
```

```
// const vs constexpr:
```

- constexpr гарантирует вычисление при компиляции
- const может вычисляться во время выполнения

```
// constexpr: инициализация при компиляции (C++20)
constexpr int global_var = 42;
```

```
// mutable: может изменяться в const объектах
mutable int cache_size;    // Для кеширования в const методах
```

```
// volatile: запрещает оптимизацию компилятора
volatile int hardware_reg; // Для работы с железом
```

5. Арифметические операторы

Бинарные операторы:

- + Сложение: $a + b$
- - Вычитание: $a - b$
- * Умножение: $a * b$
- / Деление: a / b
- % Остаток от деления: $a \% b$ (только для целых)

Унарные операторы:

- + Унарный плюс: $+a$
- - Унарный минус: $-a$

Примеры:

```
int a = 10, b = 3;
```

```
int sum = a + b;    // 13
```

```
int diff = a - b;   // 7
```

```
int product = a * b; // 30
```

```
int quotient = a / b; // 3 (целочисленное деление)
```

```
int remainder = a % b; // 1
```

```
double x = 10.0, y = 3.0;
```

```
double div = x / y;    // 3.333...
```

6. Операторы присваивания

// Простое присваивание:

```
int a = 5;
```

```
a = 10;           // Теперь a = 10
```

// Составные операторы присваивания:

```
a += 3;           // Эквивалентно: a = a + 3
```

```
a -= 2;           // a = a - 2
```

```
a *= 4;           // a = a * 4
```

```
a /= 2;           // a = a / 2
```

```
a %= 3;           // a = a % 3
```

// Побитовые составные операторы:

```
a &= mask;        // a = a & mask
```

```
a |= flags;       // a = a | flags
```

```
a ^= value;       // a = a ^ value
```

```
a <<= 2;           // a = a << 2
```

```
a >>= 1;           // a = a >> 1
```

// Преимущества составных операторов:

- Более компактная запись
- Лучшая производительность (иногда)
- Удобство для сложных выражений

7. Инкремент и декремент

// Постфиксные операторы (возвращают старое значение):

```
int a = 5;  
int b = a++;    // b = 5, a = 6  
int c = a--;    // c = 6, a = 5
```

// Префиксные операторы (возвращают новое значение):

```
int d = ++a;    // a = 6, d = 6  
int e = --a;    // a = 5, e = 5
```

// Разница в использовании:

```
int arr[] = {1, 2, 3, 4, 5};  
int i = 0;
```

// Постфиксный - использовать значение, затем увеличить

```
int value = arr[i++]; // value = arr[0], i = 1
```

// Префиксный - увеличить, затем использовать значение

```
int next = arr[++i];  // i = 2, next = arr[2]
```

// Рекомендация MIT:

Используйте префиксную форму если не нужен результат операции

8. Операторы сравнения

```
// Операторы возвращают bool (true/false):
```

```
int a = 5, b = 3;
```

```
bool eq = (a == b);    // Равно: false
```

```
bool ne = (a != b);    // Не равно: true
```

```
bool gt = (a > b);     // Больше: true
```

```
bool lt = (a < b);     // Меньше: false
```

```
bool ge = (a >= b);    // Больше или равно: true
```

```
bool le = (a <= b);    // Меньше или равно: false
```

```
// Особенности сравнения вещественных чисел:
```

```
double x = 0.1 + 0.2;
```

```
double y = 0.3;
```

```
// НЕПРАВИЛЬНО (из-за ошибок округления):
```

```
bool wrong = (x == y); // Может быть false!
```

```
// ПРАВИЛЬНО (с допуском):
```

```
const double EPSILON = 1e-10;
```

```
bool correct = std::abs(x - y) < EPSILON; // true
```

```
// Трехстороннее сравнение (C++20):
```

```
auto cmp = (a <=> b); // Возвращает std::strong_ordering
```

9. Логические операторы

```
// Логическое И: && (возвращает true если оба true)
bool a = true, b = false;
bool result1 = a && b;    // false
bool result2 = a && true; // true

// Логическое ИЛИ: || (возвращает true если хотя бы один true)
bool result3 = a || b;    // true
bool result4 = false || b; // false

// Логическое НЕ: ! (инверсия)
bool result5 = !a;        // false
bool result6 = !b;        // true

// Короткое замыкание (short-circuit evaluation):
// Правая часть не вычисляется если результат ясен из левой

if (ptr != nullptr && ptr->is_valid()) {
    // Если ptr == nullptr, ptr->is_valid() не вызывается
}

if (value < 0 || value > 100) {
    // Если value < 0, value > 100 не вычисляется
}

// Побитовые логические операторы (работают с битами):
// & (И), | (ИЛИ), ^ (XOR), ~ (NOT)
```

10. Побитовые операторы

```
// Работают с отдельными битами чисел:
unsigned char a = 0b0011'0011; // 51
unsigned char b = 0b0000'1111; // 15

// Побитовое И: &
unsigned char and_result = a & b; // 0b0000'0011 (3)

// Побитовое ИЛИ: |
unsigned char or_result = a | b; // 0b0011'1111 (63)

// Побитовое XOR: ^
unsigned char xor_result = a ^ b; // 0b0011'1100 (60)

// Побитовое НЕ: ~
unsigned char not_result = ~a; // 0b1100'1100 (204)

// Сдвиги:
unsigned char c = 0b0000'1100; // 12

// Сдвиг влево: <<
unsigned char left_shift = c << 2; // 0b0011'0000 (48)

// Сдвиг вправо: >>
unsigned char right_shift = c >> 2; // 0b0000'0011 (3)

// Применение: флаги, маски, низкоуровневые операции
```

11. Приоритет операторов

Высший приоритет (выполняются первыми):

1. :: Scope resolution
2. () Function call
3. [] Array subscript
4. . -> Member access
5. ++ -- Postfix increment/decrement

Арифметические:

6. ++ -- Prefix increment/decrement
7. + - Unary plus/minus
8. * / % Multiplication/division/modulus
9. + - Addition/subtraction

Сравнение и логика:

10. < > <= >= Comparison
11. == != Equality
12. && Logical AND
13. || Logical OR

Низший приоритет:

14. = += -= Assignment

// Всегда используйте скобки для ясности!

`int result = (a + b) * c; // Ясно`

`int unclear = a + b * c; // Неясно: b*c или a+b сначала?`

12. Преобразование типов

```
// Неявное преобразование (автоматическое):  
int i = 5;  
double d = i;          // int → double (расширение)  
  
float f = 3.14f;  
int j = f;             // float → int (сужение, потеря данных!)  
  
// Явное преобразование (C-style):  
double pi = 3.14159;  
int approx = (int)pi;  // 3 (отбрасывание дробной части)  
  
// static_cast (modern C++ - РЕКОМЕНДУЕТСЯ):  
int value = 100;  
double precise = static_cast<double>(value);  
  
// const_cast: снятие/добавление const  
const int* ptr = &value;  
int* mutable_ptr = const_cast<int*>(ptr);  
  
// reinterpret_cast: низкоуровневое преобразование  
int num = 65;  
char* char_ptr = reinterpret_cast<char*>(&num);  
  
// dynamic_cast: преобразование в иерархии классов
```

13. Ключевое слово auto (C++11)

```
// auto: компилятор выводит тип автоматически
auto x = 5;           // int
auto y = 3.14;        // double
auto z = 'A';         // char
auto flag = true;     // bool

// Полезно для сложных типов:
std::vector<std::string> names = {"Alice", "Bob"};
auto it = names.begin(); // std::vector<std::string>::iterator

// auto с ссылками:
int value = 42;
auto& ref = value;      // int&
auto copy = value;      // int (копия)

// decltype: получение типа выражения
decltype(x) another_x = x; // Тип такой же как у x

// auto в циклах:
for (auto& name : names) {
    // name имеет тип std::string&
    std::cout << name << std::endl;
}

// Рекомендация MIT:
Используйте auto для улучшения читаемости сложных типов
```

14. Область видимости переменных

```
// Локальная область видимости (внутри блоков {}):  
void function() {  
    int local_var = 10; // Видна только внутри function()  
    if (true) {  
        int block_var = 20; // Видна только внутри if  
        local_var = block_var;  
    }  
    // block_var не видна здесь!  
}  
  
// Глобальная область видимости:  
int global_var = 100; // Видна во всех функциях  
  
// Область видимости пространства имен:  
namespace Math {  
    const double PI = 3.14159; // Math::PI  
}  
  
// Статические переменные:  
void counter() {  
    static int count = 0; // Сохраняет значение между вызовами  
    count++;  
}  
  
// Время жизни:  
• Автоматическое: уничтожаются при выходе из области видимости  
• Статическое: существуют всю программу  
• Динамическое: управляется вручную (new/delete)
```

15. Лучшие практики: Именование

Стандарты именования:

// snake_case для переменных и функций:

```
int student_count;
```

```
float average_grade;
```

```
void calculate_statistics();
```

// UPPER_CASE для констант:

```
const int MAX_STUDENTS = 100;
```

```
const double PI_VALUE = 3.14159;
```

// Осмысленные имена:

ПЛОХО: int x, int temp, int var1

ХОРОШО: int user_age, int item_count, int total_score

// Правило одной ответственности:

- Одна переменная - одна концепция
- Избегайте многоцелевых переменных

// Инициализация при объявлении:

```
int count = 0;    // ХОРОШО
```

```
int count;        // ПЛОХО (неинициализированная)
```

```
count = 0;        // лучше инициализировать сразу
```

// const везде где возможно:

```
const int immutable_value = 42;
```

```
const double calculated_result = compute_value();
```

16. Лучшие факторы: Выбор типов

// Используйте подходящий размер типа:

- `int`: для целых чисел (по умолчанию)
- `size_t`: для размеров и индексов (unsigned)
- `double`: для вычислений с плавающей точкой
- `float`: только если критична память

// Фиксированный размер (cstdint):

```
#include <cstdint>
```

```
int8_t small;    // 8-bit signed
```

```
uint16_t medium; // 16-bit unsigned
```

```
int32_t large;   // 32-bit signed
```

```
uint64_t huge;   // 64-bit unsigned
```

// Беззнаковые типы:

- Используйте для величин, которые не могут быть отрицательными
- Осторожно с арифметикой (переполнение, сравнение)

// Современные типы C++:

- `std::byte`: для сырых данных (C++17)
- `std::string_view`: для строковых параметров (C++17)
- `std::optional`: для опциональных значений (C++17)

17. Best practices: Операции

// Избегайте магических чисел:

```
const int MAX_RETRIES = 3;    // ХОРОШО  
if (attempts < 3) { }        // ПЛОХО
```

// Скобки для ясности:

```
int result = (a + b) * c;    // Ясно  
int unclear = a + b * c;    // Неясно
```

// Проверка деления на ноль:

```
if (divisor != 0) {  
    result = dividend / divisor;  
} else {  
    // Обработка ошибки  
}
```

// Избегайте сложных выражений:

// ПЛОХО:

```
int value = a + b * c - d / e + f % g;
```

18. Распространенные ошибки

1. Неинициализированные переменные:

```
int count;      // Мусорное значение
std::cout << count; // Неопределенное поведение
```

2. Переполнение числовых типов:

```
int max_int = 2147483647;
max_int += 1;    // Переполнение -> -2147483648
```

3. Потеря точности при преобразованиях:

```
double pi = 3.14159;
int approx = pi;  // 3 (потеря дробной части)
```

4. Сравнение вещественных чисел через ==:

```
double a = 0.1 + 0.2;
double b = 0.3;
if (a == b) { }    // Ложное отрицание
```

5. Путаница с операторами:

```
if (x = 5) { }    // Присваивание вместо сравнения
```

6. Игнорирование возвращаемых значений:

```
std::cin >> value; // Может fail
if (std::cin.fail()) { /* обработка ошибки */ }
```

7. Выход за границы массива:

```
int arr[5];
arr[5] = 10;    // Неопределенное поведение
```

19. Отладка и диагностика

// Вывод значений для отладки:

```
std::cout << "x = " << x << ", y = " << y << std::endl;
```

// Статические анализаторы:

- -Wall -Wextra -pedantic в GCC/Clang
- /W4 в MSVC
- Статический анализ кода

// Инструменты отладки:

- GDB/LLDB: print variable, watch variable
- IDE debuggers: точки останова, наблюдение
- Sanitizers: AddressSanitizer, UndefinedBehaviorSanitizer

// Логирование:

```
#define DEBUG_LOG(msg) std::cout << __LINE__ << ": " << msg << std::endl
```

// Проверка предположений:

```
#include <cassert>
```

```
assert(x >= 0 && "x must be non-negative");
```

// Обработка ошибок ввода:

```
int value;
```

```
if (!(std::cin >> value)) {
```

```
    std::cerr << "Invalid input!" << std::endl;
```

```
    std::cin.clear(); // Сброс флагов ошибок
```

```
    std::cin.ignore(); // Очистка буфера
```

```
}
```

20. Практические примеры

```
// Конвертер температур:
double celsius_to_fahrenheit(double celsius) {
    return (celsius * 9.0 / 5.0) + 32.0;
}
```

```
// Вычисление площади:
double circle_area(double radius) {
    const double PI = 3.14159;
    return PI * radius * radius;
}
```

```
// Обмен значений:
void swap_values(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

}
```

```
// Подсчет статистики:
void calculate_stats(const
std::vector<int>& data) {
    int sum = 0;
    int min = data[0];
    int max = data[0];

    for (int value : data) {
        sum += value;
        if (value < min) min = value;
        if (value > max) max = value;
    }

    double average =
static_cast<double>(sum) / data.size();
    std::cout << "Sum: " << sum << ",
Average: " << average << std::endl;
```

21. Вопросы для самопроверки MIT

1. В чем разница между `int` и `double`?
2. Когда использовать `constexpr` вместо `const`?
3. Почему `{}` инициализация предпочтительнее?
4. Как работает короткое замыкание в логических операторах?
5. В чем разница между `++i` и `i++`?
6. Как безопасно сравнивать вещественные числа?
7. Когда использовать `auto`?
8. Какие преимущества у составных операторов присваивания?
9. Как избежать переполнения числовых типов?
10. Какие инструменты отладки вы знаете?

Практические задания:

1. Реализуйте калькулятор BMI
2. Напишите конвертер валют
3. Создайте программу для статистического анализа данных

22. Современные возможности C++

```
// Structured bindings (C++17):
auto [min, max] = std::minmax({5, 2, 8, 1});

// if/switch с инициализацией (C++17):
if (int result = compute(); result > 0) {
    // Использование result
}

// constexpr if (C++17):
if constexpr (sizeof(int) == 4) {
    // Код для 32-битных систем
}

// Spaceship operator <=> (C++20):
auto compare = (a <=> b);
if (compare < 0) { /* a < b */ }

// Concepts (C++20):
template<std::integral T>
T square(T x) { return x * x; }

// Modules (C++20):
import math; // Вместо #include <cmath>

// Ranges (C++20):
auto even_numbers = numbers | std::views::filter([](int n) {
    return n % 2 == 0;
});
```

23. Заключение и ключевые выводы

- Переменные и операторы - фундамент программирования на C++
- Правильный выбор типов критически важен для производительности
- Современный C++ предлагает безопасные способы инициализации
- Понимание приоритета операторов предотвращает ошибки
- `const` и `constexpr` улучшают безопасность и производительность
- Современные возможности (`auto`, `structured bindings`) упрощают код

Ресурсы для изучения

Книги:

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

Онлайн ресурсы:

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

Статьи и руководства:

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