



Язык C++. Типы данных и константы

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

План презентации

1. Прimitивные типы данных
2. Составные типы данных
3. Пользовательские типы данных
4. Статическая типизация
5. Динамическая типизация
6. Приведение типов
7. Константы и `const` keyword
8. `constexpr` (C++11)
9. Ввод и вывод данных
10. Пространства имен
11. Псевдонимы типов



Что такое типы данных?

Определение: Тип данных определяет характеристики переменной

Ключевые аспекты:

- Размер памяти для хранения
- Диапазон допустимых значений
- Допустимые операции
- Способ представления в памяти

Категории типов в C++:

1. Примитивные (встроенные) типы
2. Составные типы
3. Пользовательские типы

Система типов C++ обеспечивает статическую типизацию, безопасность типов и поддержку пользовательских типов.

1. Примитивные типы данных

Целочисленные типы:

- char - символьный тип (1 байт)
- short - короткое целое (2 байта)
- int - целое число (4 байта)
- long - длинное целое (4-8 байт)
- long long - очень длинное целое (8 байт)

Вещественные типы:

- float - одинарная точность (4 байта)
- double - двойная точность (8 байт)
- long double - расширенная точность (8-16 байт)

Логический тип:

- bool - логическое значение (1 байт)

Специальный тип:

- void - отсутствие типа

Примитивные типы - фундаментальные строительные блоки C++ программ

Целочисленные типы

// Размеры и диапазоны (зависят от платформы)

char c = 'A'; // обычно 1 byte, -128 to 127 или 0 to 255

signed char sc = -10; // гарантированно знаковый

unsigned char uc = 200; // гарантированно беззнаковый

short s = 1000; // обычно 2 bytes, -32768 to 32767

unsigned short us = 40000; // 0 to 65535

int i = 100000; // обычно 4 bytes, -2^{31} to $2^{31}-1$

unsigned int ui = 4000000000; // 0 to $2^{32}-1$

long l = 1000000L; // 4 или 8 bytes

long long ll = 10000000000LL; // 8 bytes, -2^{63} to $2^{63}-1$

// Литералы

int decimal = 42;

int octal = 052; // начинается с 0

int hexadecimal = 0x2A; // начинается с 0x

int binary = 0b101010; // C++14, начинается с 0b

Вещественные типы

```
// Точность и диапазоны
float f = 3.14f;           // 4 bytes, ~7 значащих цифр
double d = 3.1415926535; // 8 bytes, ~15 значащих цифр
long double ld = 3.14159265358979323846L; // 8-16 bytes
```

```
// Специальные значения
float inf = 1.0f / 0.0f;    // infinity
float nan = 0.0f / 0.0f;    // not-a-number
```

```
// Литералы
double normal = 123.456;
double scientific = 1.23456e2; // 123.456
```

```
// Особенности вычислений
double a = 0.1;
double b = 0.2;
double sum = a + b;    // не точно 0.3!
bool equal = (sum == 0.3); // false!
```

```
// Правильное сравнение
bool almost_equal = std::abs(sum - 0.3) < 1e-9;
```

void и bool типы

```
// void тип
// Используется для:
// 1. Функций без возвращаемого значения
void print_hello() {
    cout << "Hello!" << endl;
}

// 2. Указателей на любой тип
void* generic_pointer;
int x = 10;
generic_pointer = &x;

// bool тип
bool is_ready = true;
bool is_empty = false;

// Преобразования
bool b1 = 10;           // true (ненулевое значение)
bool b2 = 0;            // false
bool b3 = 3.14;         // true
bool b4 = "hello";      // true
bool b5 = nullptr;     // false

// Обратное преобразование
int num = true;         // 1
int zero = false;      // 0
```

Модификаторы типов

```
// Модификаторы знака
signed int s_val = -10;    // знаковое (по умолчанию для int)
unsigned int u_val = 10;   // беззнаковое
```

```
// Модификаторы размера
short int small = 100;     // 2 bytes
long int big = 1000000L;   // 4 или 8 bytes
long long int huge = 10000000000LL; // 8 bytes
```

```
// Комбинации модификаторов
unsigned short us = 65535;
unsigned long ul = 4000000000UL;
unsigned long long ull = 18446744073709551615ULL;
```

```
// Сокращенные формы
short s = 100;             // вместо short int
unsigned u = 50000;        // вместо unsigned int
long long ll = 9999999999LL; // вместо long long int
```

```
// Рекомендации:
// • Используйте unsigned для битовых операций
// • Будьте осторожны с mixed signed/unsigned сравнениями
// • Используйте size_t для размеров и индексов
```

2. Составные типы данных

Составные типы строятся из примитивных:

1. Массивы (Arrays)

- Однородная коллекция элементов
- Фиксированный размер
- Быстрый доступ по индексу

2. Указатели (Pointers)

- Хранят адрес памяти
- Могут указывать на любой тип
- Используются для динамической памяти

3. Ссылки (References)

- Псевдоним для существующей переменной
- Должны быть инициализированы при объявлении
- Не могут быть перенаправлены

4. Структуры (Structs)

- Гетерогенная коллекция данных
- Могут содержать разные типы

Составные типы позволяют создавать сложные структуры данных

Массивы

```
// Статические массивы
int numbers[5];           // объявление
int primes[5] = {2, 3, 5, 7, 11}; // инициализация
int auto_size[] = {1, 2, 3}; // автоматический размер

// Доступ к элементам
primes[0] = 13;           // изменяем первый элемент
int first = primes[0];    // читаем первый элемент
int size = sizeof(primes) / sizeof(primes[0]); // размер массива

// Многомерные массивы
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
int value = matrix[1][2]; // 6

// C-style strings (массивы символов)
char name[20] = "John";   // автоматически добавляется '\0'
char greeting[] = "Hello"; // размер 6 (5 символов + '\0')

// std::array (C++11, рекомендуется)
#include <array>
std::array<int, 5> arr = {1, 2, 3, 4, 5};
int size = arr.size();    // 5
int first = arr.at(0);    // безопасный доступ
```

Указатели

```
// Базовые операции с указателями
int x = 10;
int* ptr = &x;          // ptr хранит адрес x
cout << *ptr;           // разыменование: 10

// Изменение через указатель
*ptr = 20;               // x теперь = 20

// Указатель на указатель
int** pptr = &ptr;
**pptr = 30;             // x теперь = 30

// Указатель на константу
const int* cptr = &x;    // можно читать,但不能修改
int value = *cptr;       // OK
// *cptr = 40;           // ошибка!

// Константный указатель
int* const const_ptr = &x; // указатель нельзя изменить
*const_ptr = 40;          // OK
// const_ptr = &y;        // ошибка!

// Нулевой указатель
int* null_ptr = nullptr;  // современный C++
int* old_null = NULL;     // устаревший стиль
```

Ссылки

```
// Ссылки как псевдонимы
int x = 10;
int& ref = x;           // ref - псевдоним для x
ref = 20;               // x теперь = 20
cout << x;             // 20

// Особенности ссылок:
// • Должны быть инициализированы при объявлении
// • Не могут быть перенаправлены
// • Не могут быть null
// • Синтаксически выглядят как переменные

// Константные ссылки
const int& cref = x;    // можно только читать
int value = cref;       // OK
// cref = 30;          // ошибка!

// Ссылки в функциях
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
int a = 5, b = 10;
swap(a, b);             // a=10, b=5

// R-value ссылки (C++11)
int&& rref = 42;         // ссылка на временный объект
```

3. Пользовательские типы

```
// Структуры (struct)
struct Point {
    double x;
    double y;
};
Point p1 = {1.0, 2.0};
p1.x = 3.0;

// Классы (class) - по умолчанию private доступ
class Rectangle {
private:
    double width;
    double height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double area() const { return width * height; }
};
Rectangle rect(5.0, 3.0);

// Перечисления (enum)
enum Color { RED, GREEN, BLUE };
Color c = GREEN;

// Strongly-typed enums (C++11)
enum class TrafficLight : uint8_t { RED, YELLOW, GREEN };
TrafficLight light = TrafficLight::RED;

// using для синонимов типов (C++11)
using String = std::string;
using IntVector = std::vector<int>;
```

4. Статическая типизация

C++ использует статическую типизацию:

Преимущества:

- Проверка типов на этапе компиляции
- Раннее обнаружение ошибок
- Лучшая производительность
- Самодокументирующийся код
- Безопасность типов

Особенности:

- Тип переменной определяется при объявлении
- Нельзя изменить тип после объявления
- Компилятор проверяет совместимость типов
- Требуется явное приведение типов

Пример:

```
int x = 10;      // тип int
// x = "hello"; // ошибка
                // компиляции!
double y = 3.14; // тип double
// int z = y;    // предупреждение о
                // потере точности
```

Статическая типизация обеспечивает надежность и производительность

5. Динамическая типизация

C++ не поддерживает **чистую** динамическую типизацию, но есть механизмы для работы с типами во время выполнения:

1. RTTI (Run-Time Type Information)

- typeid - получение информации о типе
- dynamic_cast - безопасное приведение типов

2. Шаблоны (Templates)

- Позволяют писать обобщенный код
- Типы определяются при инстанцировании

3. std::variant (C++17)

- Может хранить значения разных типов
- Тип-safe union

4. std::any (C++17)

- Может хранить любой тип
- Аналогичен dynamic typing

5. Виртуальные функции

- Полиморфизм во время выполнения
- Позволяют работать с разными типами через базовый интерфейс

C++ предоставляет инструменты для гибкой работы с типами

6. Приведение типов

```
// Неявное преобразование (автоматическое)
int i = 10;
double d = i;           // int → double
float f = 3.14;
int j = f;              // float → int (потеря точности)

// C-style cast (не рекомендуется)
double price = 99.99;
int int_price = (int)price; // 99

// static_cast (безопасное приведение)
int total = 123;
double double_total = static_cast<double>(total);

// const_cast (удаление const)
const int ci = 10;
int* modifiable = const_cast<int*>(&ci);

// reinterpret_cast (низкоуровневое приведение)
int number = 65;
char* as_char = reinterpret_cast<char*>(&number);

// dynamic_cast (безопасное приведение для полиморфизма)
Base* base = new Derived();
Derived* derived = dynamic_cast<Derived*>(base);
if (derived) { /* успешное приведение */ }
```

7. Константы

1. Литеральные константы:

- 42, 3.14, 'A', "hello", true

2. Константы препроцессора:

- #define MAX_SIZE 100
- #define PI 3.14159

3. const переменные:

- const int MAX = 100;
- const double PI = 3.14159;

4. constexpr (C++11):

- constexpr int SIZE = 100;
- Вычисляются на этапе компиляции

5. Перечисления:

- enum { RED, GREEN, BLUE };
- enum class Color { RED, GREEN, BLUE };

Преимущества const над #define:

- Проверка типов
- Область видимости
- Отладка
- Безопасность

Ключевое слово const

```
// Константные переменные
const int MAX_SIZE = 100;
const double PI = 3.14159;

// Указатели и const
const int* ptr1;           // указатель на константу
int* const ptr2;           // константный указатель
const int* const ptr3;     // константный указатель на константу

// Константные ссылки
int x = 10;
const int& ref = x;        // нельзя изменить через ref

// Константные методы
class Circle {
    double radius;
public:
    double get_area() const { // не изменяет состояние
        return 3.14 * radius * radius;
    }
    void set_radius(double r) { // изменяет состояние
        radius = r;
    }
};

// Константные объекты
const Circle unit_circle(1.0);
double area = unit_circle.get_area(); // OK
// unit_circle.set_radius(2.0);        // ошибка!
```

8. constexpr (C++11)

```
// constexpr переменные
constexpr int SIZE = 100;           // вычисляется на этапе компиляции
constexpr double PI = 3.1415926535;

// constexpr функции
constexpr int square(int x) {
    return x * x;
}
constexpr int squared = square(5); // 25, вычисляется на этапе компиляции

// constexpr с условиями (C++14)
constexpr int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
constexpr int fact_5 = factorial(5); // 120

// constexpr с циклами (C++14)
constexpr int sum_up_to(int n) {
    int result = 0;
    for (int i = 1; i <= n; ++i) {
        result += i;
    }
    return result;
}
constexpr int sum_10 = sum_up_to(10); // 55

// Преимущества constexpr:
// • Вычисления на этапе компиляции
// • Нет накладных расходов во время выполнения
// • Безопасность типов
```

9. Ввод и вывод данных

```
// Стандартный ввод/вывод
#include <iostream>
using namespace std;

// Вывод данных
cout << "Hello, World!" << endl;
cout << "Number: " << 42 << endl;
cout << "Price: $" << 99.99 << endl;

// Ввод данных
int age;
cout << "Enter your age: ";
cin >> age;

string name;
cout << "Enter your name: ";
cin >> name;           // чтение до пробела
getline(cin, name);     // чтение всей строки

// Форматированный вывод
#include <iomanip>
cout << fixed << setprecision(2);
cout << "Price: $" << 99.999 << endl; // $100.00

// Файловый ввод/вывод
#include <fstream>
ofstream outfile("data.txt");
outfile << "Some data" << endl;
outfile.close();
```

10. Пространства имен

```
// Определение namespace
namespace math {
    const double PI = 3.14159;

    double square(double x) {
        return x * x;
    }

    namespace geometry {
        double circle_area(double radius) {
            return PI * square(radius);
        }
    }
}

// Использование с квалификатором
double area = math::geometry::circle_area(5.0);

// using declaration
using math::PI;
double circumference = 2 * PI * 5.0;

// using directive (осторожно!)
using namespace math;
double squared = square(4.0);

// Анонимный namespace (только в текущем файле)
namespace {
    int internal_variable = 42;
}

// std namespace
using std::cout;
using std::endl;
using std::string;
```

11. Псевдонимы типов

```
// typedef (устаревший, но поддерживается)
typedef unsigned int uint;
typedef int* int_ptr;
typedef double (*MathFunc)(double); // указатель на функцию
```

```
// using (C++11, рекомендуется)
using uint = unsigned int;
using int_ptr = int*;
using MathFunc = double (*)(double);
```

```
// Псевдонимы для сложных типов
using StringVector = std::vector<std::string>;
using IntMatrix = std::vector<std::vector<int>>>;
using Callback = void (*)(int, const std::string&);
```

```
// Шаблонные псевдонимы (C++11)
template<typename T>
using Pointer = T*;
Pointer<int> int_ptr; // int*
```

```
template<typename T>
using Vector = std::vector<T>;
Vector<std::string> names;
```

```
// auto (C++11) - автоматическое определение типа
auto x = 10;           // int
auto name = "John";    // const char*
auto& ref = x;         // int&
const auto pi = 3.14;  // const double
```

Лучшие практики (Best Practices)

Рекомендации по работе с типами:

1. Используйте `auto` для очевидных типов
2. Предпочитайте `using` вместо `typedef`
3. Используйте `const` везде, где возможно
4. Используйте `constexpr` для compile-time констант
5. Избегайте C-style cast, используйте `static_cast`
6. Используйте `nullptr` вместо `NULL` или `0`
7. Выбирайте подходящий целочисленный тип
8. Используйте `size_t` для размеров и индексов
9. Избегайте неявных преобразований типов
10. Используйте brace initialization `{}`

Рекомендации по константам:

- Используйте `const` вместо `#define`
- Используйте `constexpr` для вычислений на этапе компиляции
- Используйте `enum class` вместо обычных `enum`
- Используйте константные ссылки для параметров функций

Следование лучшим практикам улучшает качество и надежность кода.

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++



Заключение

Ключевые выводы:

- Система типов C++ обеспечивает безопасность и производительность
- Понимание типов данных критически важно для эффективного программирования
- Современный C++ предлагает мощные инструменты для работы с типами
- Правильное использование `const` и `constexpr` улучшает качество кода
- Следование `best practices` обязательно для профессиональной разработки

"Программирование на C++ - это искусство выбора правильных типов и абстракций."

Дальнейшие шаги:

- Изучение шаблонов (templates)
- Освоение STL (Standard Template Library)
- Изучение move semantics и perfect forwarding
- Практика с современными возможностями C++17/20