

Лекция 4. Представление данных. Побитовые операции. Генерация случайных чисел

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

- 1 Системы счисления: двоичная, восьмеричная, шестнадцатеричная.
- 2 Представление целых и вещественных чисел.
- 3 Побитовые операции: `&`, `|`, `^`, `~`, `<<`, `>>`.
- 4 Битовые маски: установка, сброс, проверка, переключение битов.
- 5 Практические применения побитовых операций.
- 6 Генерация случайных чисел в C++: `<random>`.
- 7 Сравнение с устаревшим `rand()`.

Зачем нужно понимать представление данных?

- Данные в памяти компьютера хранятся в двоичном виде.
- Понимание представления помогает:
 - работать с памятью на низком уровне;
 - понимать побитовые операции;
 - отлаживать сложные ошибки;
 - оптимизировать код.
- Побитовые операции широко применяются в:
 - системном программировании;
 - работе с аппаратурой;
 - криптографии;
 - сжатии данных;
 - флагах и битовых масках.

- Основание: 2.
- Цифры: 0 и 1.
- Пример: $1010_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 0 + 2 + 0 = 10_{10}$.
- В C++ можно записывать двоичные литералы с префиксом 0b (C++14):

```
int a = 0b1010;    // 10
int b = 0b11111111; // 255
int c = 0b10101010; // 170
```

- Разделители разрядов (C++14):

```
int d = 0b1010'1010'1010; // 2730
```

Шестнадцатеричная система

- Основание: 16.
- Цифры: 0–9, A–F.
- Компактная запись для двоичных данных.
- В C++ записывается с префиксом 0x:

```
int a = 0x1A;    // 26
int b = 0xFF;    // 255
int c = 0xDEAD;  // 57005
auto d = 0xDEAD'BEEFu; // беззнаковый литерал с разделителем, unsigned int
```

- Каждая шестнадцатеричная цифра = 4 бита.
- Удобно для отладки (дамп памяти).

Восьмеричная система (обзорно)

- Основание: 8.
- Цифры: 0–7.
- В C++ записывается с ведущим нулём:

```
int a = 012;    // 10 (восьмеричное)
```

Предупреждение

Восьмеричные литералы могут привести к путанице. Например, 012 — это 10, а не 12.

- Рекомендуется избегать восьмеричных литералов в современном коде.

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    int a = 0b1010;
    int b = 0x0A;
    int c = 012;
    cout << a << " " << b << " " << c;
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    int a = 0b1010;
    int b = 0x0A;
    int c = 012;
    cout << a << " " << b << " " << c;
    return 0;
}
```

Ответ: 10 10 10 (все три записи обозначают число 10)

Представление целых чисел

- Для отрицательных чисел используется **дополнительный код**. Дополнительный код гарантирован начиная с C++20; в C++14 допускались другие представления.
- Это позволяет использовать одно и то же аппаратное сложение для знаковых и беззнаковых чисел.
- Диапазоны:
 - `int`: от -2^{31} до $2^{31} - 1$ (обычно).
 - `unsigned int`: от 0 до $2^{32} - 1$.
- **Переполнение знаковой арифметики** — неопределённое поведение (UB). Беззнаковая арифметика выполняется по модулю 2^N .
- Пример переполнения:

```
int max = std::numeric_limits<int>::max(); // 2147483647
int overflow = max + 1; // UB!
```

Представление вещественных чисел (кратко)

- Стандарт IEEE 754:
 - float — 32 бита (1 знак, 8 экспонента, 23 мантисса).
 - double — 64 бита (1 знак, 11 экспонента, 52 мантисса).
- Из-за конечной точности:
 - числа представляются приближённо;
 - результаты приближённых вычислений могут не совпадать при сравнении через ==.
- Один из способов сравнения (не универсальный):

```
(std::abs(a - b) < std::numeric_limits<double>::epsilon())
```

`epsilon()` возвращает разность между 1 и наименьшим значением, превышающим 1, которое может быть представлено для данного типа данных.

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    double a = 0.1 + 0.2;
    double b = 0.3;
    cout << (a == b);
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    double a = 0.1 + 0.2;
    double b = 0.3;
    cout << (a == b);
    return 0;
}
```

Ответ: 0 (false). Из-за ошибок округления $0.1 + 0.2 \neq 0.3$.

Побитовые операции — введение

- Работают с отдельными битами чисел.
- Применяются к целым типам (`int`, `char`, `unsigned`, ...).
- Операции:

Оператор	Название	Пример
<code>&</code>	побитовое И	<code>a & b</code>
<code> </code>	побитовое ИЛИ	<code>a b</code>
<code>^</code>	исключающее ИЛИ	<code>a ^ b</code>
<code>~</code>	побитовая инверсия	<code>~a</code>
<code><<</code>	сдвиг влево	<code>a << n</code>
<code>>></code>	сдвиг вправо	<code>a >> n</code>

Побитовое И (&)

- Результат = 1, если оба бита равны 1.
- Используется для проверки битов.

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

```
int a = 0b1010; // 10
int b = 0b1100; // 12
int c = a & b;   // 0b1000 = 8
```

Побитовое ИЛИ (|)

- Результат = 1, если хотя бы один бит равен 1.
- Используется для установки битов.

a	b	a b
0	0	0
0	1	1
1	0	1
1	1	1

```
int a = 0b1010; // 10
int b = 0b1100; // 12
int c = a | b;   // 0b1110 = 14
```

Исключающее ИЛИ (^)

- Результат = 1, если биты разные.
- Используется для переключения битов.

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

```
int a = 0b1010; // 10
int b = 0b1100; // 12
int c = a ^ b;   // 0b0110 = 6
```

Побитовая инверсия ($\sim$)

- Инвертирует все биты ($0 \rightarrow 1, 1 \rightarrow 0$).

```
unsigned int a = 0b1010; // 10
unsigned int b = ~a;      // 0b11111111 11111111 11111111 11110101
int c = 0b101; // 5
int d = ~c; // -6, 11111111 11111111 11111111 11111010
```

Осторожно

В C++23 знаковые целые представлены в дополнительном коде.

В C++14 дополнительный код не был обязательным: результат зависел от представления знакового типа.

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    int a = 0b1011;
    int b = 0b1100;
    cout << (a & b) << " " << (a | b) << " " << (a ^ b);
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    int a = 0b1011;
    int b = 0b1100;
    cout << (a & b) << " " << (a | b) << " " << (a ^ b);
    return 0;
}
```

Ответ: 8 15 7 (0b1000, 0b1111, 0b0111)

Сдвиг влево (<<)

- Сдвигает биты влево на указанное количество позиций.
- Освободившиеся позиции заполняются нулями.
- В C++23 результат $x \ll n$ соответствует умножению на 2^n . Вышедшие за границу старшие биты отбрасываются.
- В C++14 левый сдвиг отрицательного знакового числа — UB. Для неотрицательного знакового числа произведение должно помещаться в соответствующий беззнаковый тип; иначе также возникает UB.

```
int a = 0b1010; // 10
int b = a << 1; // 0b10100 = 20 (10 * 2)
int c = a << 2; // 0b101000 = 40 (10 * 4)
```

Важно

Отрицательное число позиций и сдвиг на ширину типа или больше приводят к неопределённому поведению.

Сдвиг вправо (>>)

- Сдвигает биты вправо.
- Для беззнаковых: освободившиеся позиции заполняются нулями (логический сдвиг).
- Для знаковых типов правый сдвиг арифметический: знаковый бит копируется. Результат соответствует делению на 2^n с округлением вниз.

```
unsigned int a = 0b10100; // 20
unsigned int b = a >> 1;   // 0b1010 = 10 (20 / 2)
unsigned int c = a >> 2;   // 0b101 = 5 (20 / 4)
int d = -8;
int e = d >> 1; // -4 (арифметический сдвиг)
```

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    unsigned int a = 0b1000; // 8
    cout << (a << 1) << " " << (a >> 1);
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    unsigned int a = 0b1000; // 8
    cout << (a << 1) << " " << (a >> 1);
    return 0;
}
```

Ответ: 16 4 (8 << 1 = 16 (0b10000), 8 >> 1 = 4 (0b0100))

- Битовые маски используются для работы с отдельными битами.
- Маска — это число, в котором нужные биты установлены в 1.
- Основные операции с масками:
 - ❶ Установка бита: `x |= mask;`
 - ❷ Сброс бита: `x &= ~mask;`
 - ❸ Проверка бита: `if (x & mask) ...`
 - ❹ Переключение бита: `x ^= mask;`

```
unsigned int x = 0b0000; // 0
x |= (1 << 2);           // устанавливаем бит 2 в 1
// x = 0b0100 = 4
x |= (1 << 5);           // устанавливаем бит 5 в 1
// x = 0b100100 = 36
```

- $1 \ll n$ — маска с единицей в позиции n .
- $x \mid= \text{mask}$ — устанавливает биты, указанные в маске.

Сброс бита

```
unsigned int x = 0b1010; // 10
x &= ~(1 << 1);          // сбрасываем бит 1
// x = 0b1000 = 8
x &= ~(1 << 3);          // сбрасываем бит 3
// x = 0b0000 = 0
```

- $\sim(1 \ll n)$ — маска с нулём в позиции n и остальными единицами.
- $x \&= \sim\text{mask}$ — сбрасывает биты, указанные в маске.

Проверка бита

```
unsigned int x = 0b1010; // 10
if (x & (1 << 3)) {
    cout << "Bit 3 is set"; // выполнится
}
if (x & (1 << 2)) {
    cout << "Bit 2 is set"; // не выполнится
}
```

- $x \& (1 \ll n)$ — ненулевое значение, если бит установлен.
- Можно использовать в условиях и тернарных операторах.

Переключение бита

```
unsigned int x = 0b1010; // 10
x ^= (1 << 1);           // переключаем бит 1
// x = 0b1000 = 8 (был 1, стал 0)
x ^= (1 << 2);           // переключаем бит 2
// x = 0b1100 = 12 (был 0, стал 1)
```

- $x \oplus \text{mask}$ — инвертирует биты, указанные в маске.
- Если бит был 0 $\rightarrow$ становится 1, если 1 $\rightarrow$ становится 0.

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    unsigned int x = 0b1000;
    x |= (1 << 1);
    x &= ~(1 << 3);
    cout << x;
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    unsigned int x = 0b1000;
    x |= (1 << 1);
    x &= ~(1 << 3);
    cout << x;
    return 0;
}
```

Ответ: 2 (0b0010) — установили бит 1, сбросили бит 3

Практическая задача: проверка степени двойки

```
bool isPowerOfTwo(unsigned int x) {  
    return x != 0 && (x & (x - 1)) == 0;  
}  
  
int main() {  
    cout << isPowerOfTwo(8); // 1 (true)  
    cout << isPowerOfTwo(10); // 0 (false)  
    cout << isPowerOfTwo(0); // 0 (false)  
    return 0;  
}
```

- $x \& (x - 1)$ обнуляет младший установленный бит.
- Возвращает true, если число является степенью двойки.

Практическая задача: подсчёт установленных битов

```
int countBits(unsigned int x) {  
    int count = 0;  
    while (x) {  
        x &= (x - 1); // обнуляем младший установленный бит  
        count++;  
    }  
    return count;  
}  
  
int main() {  
    cout << countBits(0b1010); // 2  
    cout << countBits(0b1111); // 4  
    cout << countBits(0);      // 0  
    return 0;  
}
```

- Алгоритм Брайана Кернигана.
- Работает за $O(\text{количество установленных битов})$.

Практическая задача: циклический сдвиг

```
unsigned int rotateLeft(unsigned int x, int n) {  
    int bits = sizeof(x) * 8;  
    return (x << n) | (x >> (bits - n));  
}  
  
int main() {  
    unsigned int x = 0b1001; // 9  
    cout << rotateLeft(x, 1); // 0b10010 = 18 (сдвиг влево с переносом)  
    return 0;  
}
```

- Используется в криптографии и хешировании.
- Комбинация сдвигов и битового ИЛИ.
- $0 < n < bits$, т.к. при сдвиге на полную ширину типа имеем UB.

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    unsigned int flags = 0b1010;
    cout << ((flags & 0b1000) ? "Yes" : "No");
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
int main() {
    unsigned int flags = 0b1010;
    cout << ((flags & 0b1000) ? "Yes" : "No");
    return 0;
}
```

Ответ: Yes (бит 3 установлен)

- В C++ есть два подхода:
 - ❶ Устаревший: `rand()` и `srand()` (из C).
 - ❷ Современный: библиотека `<random>` (C++11).
- Современный подход предоставляет:
 - качественные генераторы;
 - различные распределения;
 - контроль над параметрами.
- Рекомендуется использовать `<random>`.

- Источник случайных значений, обычно используемый для инициализации генератора. Способ получения значений зависит от реализации; возможна псевдослучайная реализация.
- Может быть сравнительно медленным; обычно используется для получения начальных данных генератора.

```
#include <random>
std::random_device rd;
auto seed = rd(); // случайное число, unsigned int
```

Выбор генератора

```
#include <random>
std::random_device rd;
std::mt19937 gen(rd());
auto r1 = gen(); // случайное число от 0 до UINT_MAX, в auto будет unsigned int
auto r2 = gen(); // следующее
```

Другие генераторы (обзор)

- `std::mt19937` — Mersenne Twister 19937 (качественный, медленнее).
- `std::minstd_rand` — линейный конгруэнтный (быстрый, но менее качественный).
- `std::mt19937_64` — 64-битная версия.

```
std::mt19937 gen(rd());  
std::minstd_rand gen2(rd());
```

std::seed_seq (инициализация несколькими seed'ами)

```
#include <random>
std::random_device rd;
std::seed_seq seed{rd(), rd(), rd(), rd()};
std::mt19937 gen(seed);
```

- Улучшает качество инициализации.
- Позволяет инициализировать генератор последовательностью значений.

Равномерное распределение (целые)

```
#include <random>
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<int> dist(1, 6); // от 1 до 6
int dice = dist(gen); // число 1..6
for (int i = 0; i < 10; ++i)
    cout << dist(gen) << " ";
```

- `uniform_int_distribution` — равномерное распределение целых чисел.

Равномерное распределение (вещественные)

```
#include <random>
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<double> dist(0.0, 1.0);
double r = dist(gen); // число из [0.0, 1.0)
for (int i = 0; i < 5; ++i)
    cout << dist(gen) << " ";
```

- `uniform_real_distribution` — равномерное распределение вещественных чисел.
- Полуинтервал $[a, b)$.

Другие распределения (обзор)

- `normal_distribution` — нормальное (гауссовское) распределение.

```
std::normal_distribution<double> norm(5.0, 2.0); // среднее 5, стандартное отклонение 2
double g = norm(gen);
```

- `poisson_distribution` — распределение Пуассона.

```
std::poisson_distribution<int> pois(4); // среднее 4
int p = pois(gen);
```

- Много других распределений: биномиальное, экспоненциальное, и т.д.

Моделирование бросков кубика

```
#include <random>
#include <iostream>
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<int> dice(1, 6);

int sum = 0;
for (int i = 0; i < 100; ++i) {
    sum += dice(gen);
}
double average = sum / 100.0;
std::cout << "Average: " << average << std::endl;
// Пример вывода: Average: 3.52
```

- Моделируем 100 бросков шестигранного кубика.
- Считаем сумму и среднее арифметическое.
- Нет необходимости хранить все результаты.

Вопрос: что выведет этот код?

```
#include <random>
#include <iostream>
using namespace std;
int main() {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> dist(1, 3);
    cout << dist(gen);
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <random>
#include <iostream>
using namespace std;
int main() {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> dist(1, 3);
    cout << dist(gen);
    return 0;
}
```

Ответ: Случайное число от 1 до 3.

Сравнение с rand()

Характеристика	rand()	<random>
Качество	низкое	высокое
Диапазон	0..RAND_MAX (обычно 32767)	любой (задаётся распределением)
Управление распределением	нет (только равномерное)	множество распределений
Сложность	простая	более сложная, но гибкая

Рекомендуется использовать <random>.

Устаревший подход: rand()

```
#include <stdlib>
#include <ctime>
srand(time(nullptr)); // инициализация
int r = rand(); // число от 0 до RAND_MAX
int dice = rand() % 6 + 1; // 1..6
```

- rand() — линейный конгруэнтный генератор (зависит от реализации).
- Ограниченный диапазон.
- Низкое качество случайности.
- Не рекомендуется для серьёзных задач.

Пример с rand()

```
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;
int main() {
    srand(time(nullptr));
    for (int i = 0; i < 5; ++i)
        cout << rand() % 100 << " ";
    return 0;
}
```

- Простой, но ограниченный.
- Для современных задач используйте <random>.

Вопрос: что выведет этот код?

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main() {
    cout << rand() % 10;
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main() {
    cout << rand() % 10;
    return 0;
}
```

Ответ: Случайное число от 0 до 9 (но без `srand()` всегда будет одна и та же последовательность).

- Познакомились с системами счисления и представлением данных.
- Изучили побитовые операции:
 - $\&$, $|$, $\wedge$, $\sim$ — логические операции над битами.
 - $\ll$, $\gg$ — сдвиги.
- Научились работать с битовыми масками.
- Рассмотрели практические применения:
 - битовые флаги;
 - упаковка данных;
 - проверка степени двойки;
 - подсчёт битов;
 - циклический сдвиг.
- Изучили генерацию случайных чисел:
 - современный подход: `<random>`;
 - устаревший подход: `rand()`.

Стандарт	Нововведение
C++11	библиотека <code><random></code> , <code>random_device</code> , <code>default_random_engine</code> , распределения
C++14	бинарные литералы (<code>0b...</code>), разделители разрядов (<code>1'000'000</code>)

- Бинарные литералы и разделители упрощают работу с битовыми масками.
- `<random>` даёт качественные случайные числа.

- Для побитовых операций используйте беззнаковые типы `unsigned int` или `unsigned char`.
- Для случайных чисел старайтесь использовать `<random>`, а не `rand()`.
- Инициализируйте генератор с помощью `std::random_device`.
- Для разных распределений используйте соответствующие классы.
- При работе с битовыми масками используйте константы с явными значениями.

Что дальше?

- Следующая лекция: указатели и ссылки (углублённо), указатели на функции, чтение сложных объявлений.
- На практике: применение побитовых операций и генерации случайных чисел.
- Подготовьтесь к работе с указателями.

Вопросы?