06.04.2024

Битовые операции в C++

Спикер: Ибрагимов Булат Ленарович

Fast Track в Телеком, 2024

НТР новые телеком решения

PHOENIX EDUCATION

МФТИ

ИБРАГИМОВ БУЛАТ ЛЕНАРОВИЧ

- Преподаватель в МФТИ. Проводит курсы по C++ и алгоритмам, структурам данных
- Научный сотрудник Института Искусственного Интеллекта (AIRI)
- Работал разработчиком-исследователем в Яндекс и Сбербанк

Битовые операции в C++

Устройство целочисленных типов

Если число представляется в виде последовательности из k бит, то оно может принимать 2^k различных значений:

- от 0 до $2^k - 1$ для беззнаковых типов
- от -2^{k-1} до $2^{k-1} - 1$ для знаковых типов (дополнительный код)

Стандарт C++ не определяет однозначно ширину целочисленных типов (кроме `char`), а также количество битов в байте.

В зависимости от стандарта языка некоторые битовые операции над знаковыми типами могут иметь неопределенное поведение.

Далее для избежания неоднозначности будем использовать **беззнаковые** целые числа с фиксированной шириной (`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`) из заголовочного файла `<cstdint>`.

Мотивация

Битовые операции используются для:

- **Эффективности:** битовые операции эффективнее арифметических
- **Компактности:** позволяют хранить несколько флагов в одном числе
- **Приложений:** криптография, сжатие данных, графика, обработка изображений, сетевое программирование

Понимание и использование битовых операций расширяет возможности разработчика и позволяет писать более эффективный код.

Операции: `~` (отрицание)

Побитовое отрицание (NOT) инвертирует все биты числа

```
uint8_t x = 0b10101010;
std::cout << std::bitset<8>(~x) << '\n'; // 01010101
```

Операции: `&` (и)

Побитовое И (AND) возвращает 1 в бите, если оба операнда равны 1

```
uint8_t x = 0b10101010;
uint8_t y = 0b11001100;
std::cout << std::bitset<8>(x & y) << '\n'; // 10001000
```

Операции: `|` (или)

Побитовое ИЛИ (OR) возвращает 1 в бите, если хотя бы один из операндов равен 1

```
uint8_t x = 0b10101010;
uint8_t y = 0b11001100;
std::cout << std::bitset<8>(x | y) << '\n'; // 11101110
```

Операции: `^` (взаимоисключающее или)

Побитовое взаимоисключающее или (XOR) возвращает 1 в бите, если операнды различны

```
uint8_t x = 0b10101010;
uint8_t y = 0b11001100;
std::cout << std::bitset<8>(x ^ y) << '\n'; // 01100110
```

Операции: `<<` (сдвиг влево)

Побитовый сдвиг влево (LSHIFT) сдвигает все биты числа влево на указанное количество позиций. Освободившиеся биты заполняются нулями.

```
uint8_t x = 0b11010010;
std::cout << std::bitset<8>(x << 3) << '\n'; // 10010000
```

Операции: `>>` (сдвиг вправо)

Побитовый сдвиг вправо (RSHIFT) сдвигает все биты числа вправо на указанное количество позиций. Освободившиеся биты заполняются нулями.

```
uint8_t x = 0b11010010;
std::cout << std::bitset<8>(x >> 3) << '\n'; // 00011010
```

Практические трюки: умножение и деление на степень двойки

Побитовый сдвиг влево на k позиций эквивалентен умножению числа на 2^k:

```
uint32_t x = 10;
std::cout << (x << 3) << '\n'; // 80
```

Побитовый сдвиг вправо на k позиций эквивалентен делению числа на 2^k:

```
uint32_t x = 80;
std::cout << (x >> 3) << '\n'; // 10
```

Практические трюки: проверка четности

Последний бит числа равен 1, если число нечетное, и 0, если число четное:

```
uint32_t x = 10;
std::cout << (x & 1u) << '\n'; // 0
```

Практические трюки: установка и сброс бита

Установка i-го бита в 1:

```
uint8_t x = 0b10101010;
uint8_t mask = 1u << 4;  // 00010000
std::cout << std::bitset<8>(x | mask) << '\n'; // 10111010
```

Сброс i-го бита в 0:

```
uint8_t x = 0b10101010;
uint8_t mask = ~(1u << 3);  // 11110111
std::cout << std::bitset<8>(x & mask) << '\n'; // 10100010
```

Практические трюки: инвертирование бита

Инвертирование i-го бита:

```
uint8_t x = 0b10101010;
uint8_t mask = 1u << 2;  // 00000100
std::cout << std::bitset<8>(x ^ mask) << '\n'; // 10101110
```

Практические трюки: проверка на степень двойки

Число является степенью двойки, если у него ровно один установленный бит:

```
uint32_t x = 16;
std::cout << ((x & (x - 1)) == 0) << '\n'; // 1
```

Практические трюки: нахождение наименьшего значащего бита

Наименьший значащий бит числа равен i, если i-й бит равен 1, а все биты с младшего до $i + 1$ равны 0:

```
uint32_t x = 0b00010000;
std::cout << __builtin_ctz(x) << '\n'; // 4
```

Практические трюки: нахождение наибольшего значащего бита

Наибольший значащий бит числа равен i, если i-й бит равен 1, а все биты с более старшего до $i - 1$ равны 0:

```
uint32_t x = 0b00010000;
std::cout << __builtin_clz(x) << '\n'; // 27
```

Замечание: часто пригождается для поиска минимальной степени двойки, большей или равной числу.

Практические трюки: выставление наименьшего значащего 0 в 1

Выставление наименьшего значащего 0 в 1:

```
uint8_t x = 0b11001111;
std::cout << std::bitset<8>(x | (x + 1)) << '\n'; // 11011111
```

Практические трюки: обнуление наименьшей значащей 1

Обнуление наименьшей значащей 1:

```
uint8_t x = 0b11001111;
std::cout << std::bitset<8>(x & (x - 1)) << '\n'; // 11001110
```

Практические трюки: обмен значениями

Обмен значениями двух переменных без использования дополнительной памяти:

```
uint8_t x = 0b10101010;
uint8_t y = 0b11001100;
x ^= y;  // храним в x биты, в которых различаются x и y
y ^= x;  // инвертируем биты, в которых различаются x и y, получаем x
x ^= y;  // инвертируем биты, в которых различаются x и y, получаем y
```

Битовые флаги и маски

Битовые флаги

Битовые флаги позволяют хранить несколько флагов в одном числе.

Под флагом здесь понимается опция, которая может быть включена или выключена.

```
const uint8_t FLAG_A = 1u << 0;
const uint8_t FLAG_B = 1u << 1;
const uint8_t FLAG_C = 1u << 2;

uint8_t flags = FLAG_C | FLAG_B;  // включены флаги B и C

flags |= FLAG_A;   // включение флага A
flags ^= FLAG_B;   // переключение флага B
flags &= ~FLAG_C;  // выключение флага C

if (flags & FLAG_A) {  // проверка флага A
    std::cout << "Flag A is set\n";
}
```

Битовые маски

Битовая маска - это число, в котором установлены только определенные биты, а остальные биты равны 0.

Переменная в которой хранятся битовые флаги, является примером битовой маски.

Битовые маски: пример

Задача: дано n (< 64) элементов. Необходимо хранить подмножества данного набора элементов

Решение: поставим в соответствие каждому элементу число - его позицию в 64-битном числе

```
uint64_t set1 = ...;
uint64_t set2 = ...;

set1 & (1u << i);   // проверка наличия элемента i в множестве set1
set1 | (1u << i);   // добавление элемента i в множество set1
set1 & ~(1u << i);  // удаление элемента i из множества set1
set1 | set2;        // объединение множеств set1 и set2
set1 & set2;        // пересечение множеств set1 и set2
set1 ^ set2;        // симметрическая разность множеств set1 и set2
```

Резюме

- Битовые операции позволяют эффективно работать целыми числами на низком уровне
- Побитовые операции: `~`, `&`, `|`, `^`, `<<`, `>>`
- Практическое применение битовых операций включает в себя: флаги, маски, трюки с битами
- А еще множество полезных применений в различных областях программирования