C++ Russia 2024

Опасность устарела

неопределенность недопустима: исследуем изменения в UB в C++ 20/23/26

Сергей Талантов
Архитектор ПО,
Security champion

KasperskyOS

Содержание

Что такое UB

Почему UB - это плохо?

Почему UB - это хорошо?

UB в современных стандартах

Что в итоге

Что такое UB

Добро пожаловать в мир изменчивого поведения C++:

- Поведение, определяемое реализацией
- Неуточненное поведение
- Неопределенное поведение

Поведение, определяемое реализацией (implementation-defined behavior) – не определено стандартом, определено реализацией.

- Полный список есть в стандарте (Index of implementation-defined behavior)
- Размер указателя
- Определение макроса NULL
- Знаковость типа char
- Размер базовых типов кроме char
- …

Неуточненное поведение

(unspecified behavior) – стандарт определяет несколько вариантов.

- Порядок вычисления аргументов в вызове функции
- Порядок вычисления операндов операторов +, -, =, *, /, кроме &&, ||, ?:
- …

Неопределенное поведение
(undefined behavior) – программа не валидна, стандарт не налагает никаких требований, может произойти все что угодно.

- Доступ за пределами массива
- Разыменование нулевого указателя
- Целочисленное деление на ноль
- Целочисленное переполнение знаковых
- Использование памяти после освобождения
- Использование не инициализированной переменной
- Бесконечные циклы без сайд эффектов
- Гонки
- …

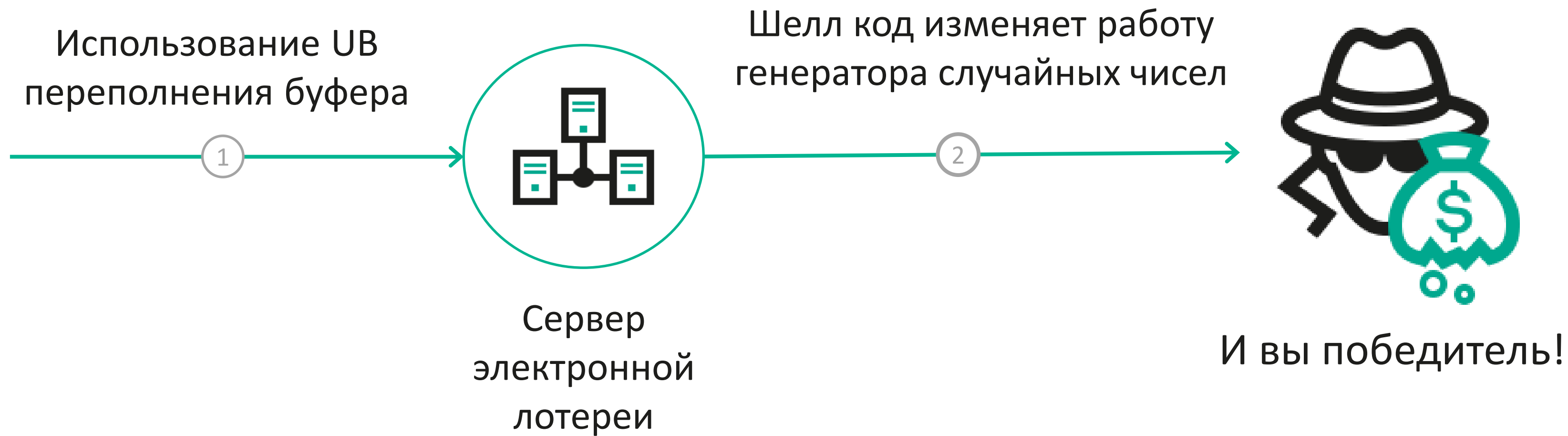
А сколько всего UB в C++?

Никто точно не знает.

- C99 J.2 Undefined behavior (193)
- P1705R1 Enumerating Core Undefined Behavior (36)
- Справочники разной степени полноты:
  https://github.com/Nekrolm/ubbook (~60)
- Правила в санитарах: clang ubsan (~35)

А реально может произойти все что угодно?

Даже неожиданный выигрыш в лотерее?



Почему UB – это плохо?

1. Проблемы с безопасностью

CWE Top 25 2023

Место	Название	Оценка	Эксплуатируемость KEV
1	Out-of-bounds Write	63.72	70
4	Use After Free	16.71	44
7	Out-of-bounds Read	14.60	2
12	NULL Pointer Dereference	6.59	0
14	Integer Overflow or Wraparound	5.89	4

2. Неожиданная оптимизация

```
int table[4];
bool exists_in_table(int v)
{
  for (int i = 0; i <= 4; i++) {
    if (table[i] == v) return true;
  }
  return false;
}
```

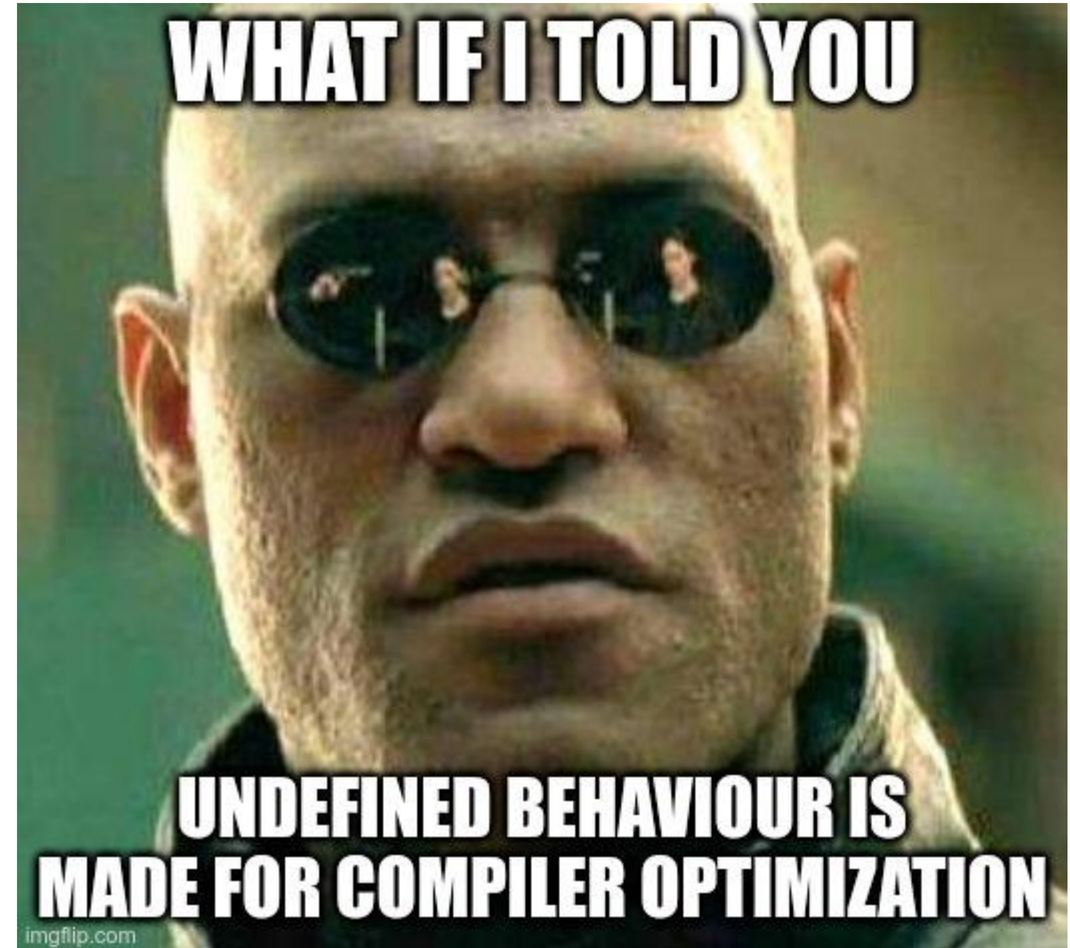
```
bool exists_in_table(int v)
{
    return true;
}
```

- Contest: Craziest Compiler Output due to Undefined Behavior
- Undefined Behavior: What Happened to My Code?
- Неопределенное поведение может привести к путешествиям во времени

Почему UB – это хорошо?

1. Скорость

- Отсутствие нулевой инициализации по-умолчанию
- Отсутствие проверок на счетчики в циклах
- Отсутствие проверок на границы буфера
- Отсутствие проверок предусловий и других ограничений
- Агрессивная оптимизация



2. Разнообразие платформ

UB в современных стандартах

C++ 20

P0907R4: Signed Integers are Two's Complement

Прямой код (sign magnitude)	Обратный код (1’s complement)	Дополнительный код (2’s complement)

	Прямой код (sign magnitude)	Обратный код (1's complement)	Дополнительный код (2's complement)
60	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0

	Прямой код (sign magnitude)	Обратный код (1’s complement)	Дополнительный код (2’s complement)
60	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0
-60	1 0 1 1 1 1 0 0	1 1 0 0 0 0 1 1	1 1 0 0 0 1 0 0

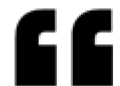
	Прямой код (sign magnitude)	Обратный код (1’s complement)	Дополнительный код (2’s complement)
60	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0
-60	1 0 1 1 1 1 0 0	1 1 0 0 0 0 1 1	1 1 0 0 0 1 0 0
0	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0

	Прямой код (sign magnitude)	Обратный код (1's complement)	Дополнительный код (2's complement)
60	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0
-60	1 0 1 1 1 1 0 0	1 1 0 0 0 0 1 1	1 1 0 0 0 1 0 0
0	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
-128	NaN	NaN	1 0 0 0 0 0 0 0

	Прямой код (sign magnitude)	Обратный код (1's complement)	Дополнительный код (2's complement)
60	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0	0 0 1 1 1 1 0 0
-60	1 0 1 1 1 1 0 0	1 1 0 0 0 0 1 1	1 1 0 0 0 1 0 0
0	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
-128	NaN	NaN	1 0 0 0 0 0 0 0

- Разное представление отрицательного числа
- Разные граничные значения
- Наличие отрицательного ноля

Ура! Теперь переполнение знаковых целых это не UB!



Note: Overflow for signed arithmetic yields undefined behavior (7.1 [expr.pre]). -- end note

Изменения в битовых сдвигах (<<, >>):

- В C++ 20 можно сдвигать отрицательные числа, раньше это было UB:

```
int x = -1 << 12;  // UB before C++ 20
```

- При сдвиге налево происходит заполнение нулями
- При сдвиге направо происходит заполнение знаковым битом
- Количество сдвигов должно быть положительным
- Количество сдвигов не должно превышать количество битов числа

C++ 20

P1152R4: Deprecating volatile

Ура! Ключевое слово, которое никто не понимает теперь выпилено!

The proposed deprecation preserves the useful parts of volatile, and removes the dubious / already broken ones.

```
volatile int x = 0;

x += 10; // C++ 20 deprecated
x++; // C++ 20 deprecated

volatile int func(volatile int arg); // C++ 20 deprecated
                                     // for non-pointers
struct Foo {int val;} bar;
volatile auto [val] = bar; // C++ 20 deprecated
```

[Deprecating volatile - JF Bastien - CppCon 2019]

C++ 20

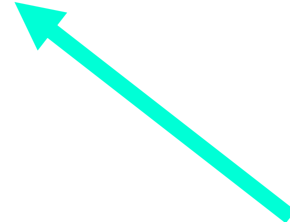
P1227: Signed ssize() functions

```
template <typename T>
bool has_repeated_values(const T& container) {
  for (int i = 0; i < container.size() - 1; ++i) {
    if (container[i] == container[i + 1])
      return true;
  }
  return false;
}
```

UB:

- Чтение за пределами массива

```
template <typename T>
bool has_repeated_values(const T& container) {
  for (int i = 0; i < std::ssize(container) - 1; ++i) {
    if (container[i] == container[i + 1])
      return true;
  }
  return false;
}
```



Теперь это не std::size_t, а std::ptrdiff_t

UB:

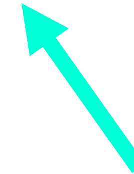
- ~~Чтение за пределами массива~~
- PTRDIFF_MAX < container.size() < SIZE_MAX

C++ 23

P2644R1: Fix for Range-based for Loop

```
std::vector<std::string> GetVector() {
  return { "str1", "str2" };
}

{
  const auto& val{ GetVector() };
  std::cout << val.front() << std::endl;  // str1
}

{
  auto&& val{ GetVector() };
  std::cout << val.front() << std::endl;  // str1
}

for (auto& val : GetVector()) {
  std::cout << val << std::endl;  // str1 str2
}
```

```
for (auto& val : GetVector().front()) {
  std::cout << val << std::endl; // UB before C++23
}
```



```
auto&& rg = GetVector().front(); // !!!
auto pos = rg.begin();
auto end = rg.end();
for ( ; pos != end; ++pos ) {
  char c = *pos;
  …
}
```

```
[&](auto&& rg) {
  auto pos = rg.begin();
  auto end = rg.end();
  for ( ; pos != end; ++pos) {
    char c = *pos;
    …
  }
}(GetVector().front());
```

C++ 23

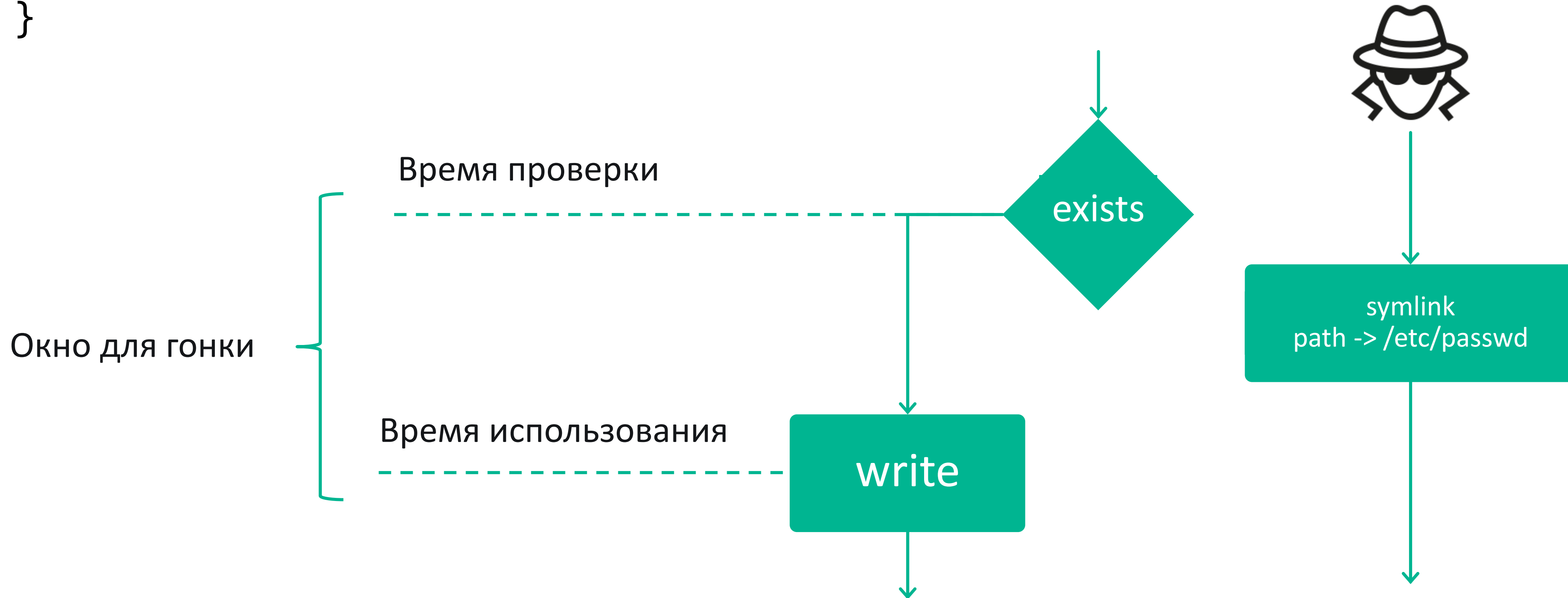
P2166R1: A Proposal to Prohibit std::basic_string and std::basic_string_view construction from nullptr.

```
basic_string( const CharT* s ); // s должна быть валидной
                                // нуль-терминальной строкой, иначе UB

std::string str{ nullptr };  // UB

basic_string( std::nullptr_t ) = delete;


std::string str{ nullptr };  // нет компиляции - нет UB

char* ch{ nullptr };
std::string str{ ch };  // UB((
```

C++ 23

P2467R1: Support exclusive mode for fstreams

```
void CheckAndCreate(const std::filesystem::path& p) {
  if (!std::filesystem::exists(p)) {
      std::fstream f(p.string(), std::ios_base::in | std::ios_base::out);
      f << "data" << std::endl;
  }
}
```



```
void CheckAndCreateNoRace(const std::filesystem::path& p)
{
  std::fstream f(p.string(),
    std::ios_base::in |
    std::ios_base::out |
    std::ios_base::noreplace);
  f << "data" << std::endl;
}
```

Все равно гонка((

```
void CheckAndUse(const std::filesystem::path& p) {
    if (std::filesystem::is_regular_file(p)) {
        std::fstream f(p.string(), std::ios_base::in | std::ios_base::out);
        f << "data";
    }
}
```

P1030R6: std::filesystem::path_view

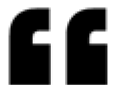
P1883R2: file_handle and mapped_file_handle

C++ 26

P2821R4: span.at()

	operator[]	at()
std::string	✓	✓
std::string_view	✓	✓
std::deque	✓	✓
std::vector	✓	✓
std::array	✓	✓
std::span	✓	✗

Почему нет в C++20, C++23? Не добавили намеренно?



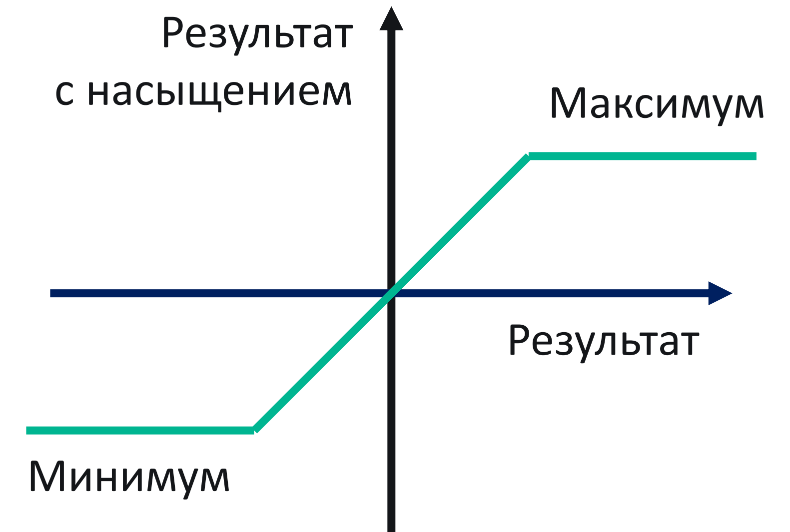
Ultimately, this becomes a stereotypical example of how C++ traditionally handles safety.

C++ 26

[P0543R3: Saturation arithmetic]

- Переполнение беззнаковых целых – циклический возврат (арифметика по модулю)
- Переполнение знаковых целых - UB

Арифметика с насыщением - это версия арифметики, в которой все операции, такие как сложение и умножение, ограничены в фиксированный диапазон между минимальным и максимальным значением.



```
template<class T>
constexpr T add_sat(T x, T y) noexcept;

template<class T>
constexpr T sub_sat(T x, T y) noexcept;

template<class T>
constexpr T mul_sat(T x, T y) noexcept;

template<class T>
constexpr T div_sat(T x, T y) noexcept;

template<class T, class U>
constexpr T saturate_cast(U x) noexcept;
```

> Most modern hardware architectures have efficient support for saturation arithmetic on SIMD vectors, including SSE2 for x86 and NEON for ARM.

Что в итоге

Выводы

- Вопросы безопасности (UB в частности) начали чаще подниматься в C++
- В современном C++ меньше шанс словить UB, чем раньше
- Процесс выпиливания UB идет не быстро

Спасибо!

Сергей Талантов

Архитектор ПО
Чемпион безопасности

Sergey.Talantov@kaspersky.com