

ТЕОРИЯ И ПРАКТИКА НАПИСАНИЯ БЕЗОПАСНОГО КОДА НА C++

ЛАПИЦКИЙ АРТЕМ

LAPITSKY.ARTEM@GMAIL.COM



WARGAMING.NET
LET'S BATTLE



ПОЧЕМУ МОЙ КОД МОЖЕТ БЫТЬ НЕБЕЗОПАСНЫМ?

С++ – это конгломерат различных языков

- C, classes, template metaprogramming, preprocessor, STL
- Множество способов решения одной задачи
- Подводные камни на стыке различных подмножеств языка

Высокая ответственность программиста

- Как можно меньше ограничивать разработчика (в том числе и в возможности выстрелить себе в ногу) – часть философии языка
- В стандарте более двухсот упоминаний ситуаций undefined behavior и около ста – unspecified behavior

ПОЧЕМУ МОЙ КОД МОЖЕТ БЫТЬ НЕБЕЗОПАСНЫМ? (2)

Возможности, которые слишком легко использовать неверно

- Адресная арифметика, перегрузка операторов, функции с переменным числом аргументов, приведение типов в стиле C...

Наследие C

- Синтаксис и философия C;
- Сочетание языка высокого уровня и максимальной эффективности;
- “You don't pay for what you don't use”;
- Небезопасные функции стандартной библиотеки.

ЗАЧЕМ ПИСАТЬ БЕЗОПАСНЫЙ КОД?



Ошибки стоят дорого

- 4 июня 1996 года на 39-й секунде полета новейшая ракета-носитель «Ariane 5» взорвалась из-за возникновения переполнения целого числа. Стоимость ракеты вместе со спутниками составила более \$8 млрд.

Время стоит дорого

- Отлаживать C/C++ программы – не самое быстрое (и приятное) занятие;
- Применение практик написания безопасного кода ускоряет процесс разработки.

ЗАЧЕМ ПИСАТЬ БЕЗОПАСНЫЙ КОД? (2)

Меньше уязвимостей

- Одна из методик разработки защищенного программного обеспечения;
- С меньшей вероятностью содержит уязвимости, которые позволят злоумышленникам совершить недружественные действия.

Выше надежность

- Обеспечение корректности работы программ;
- Путь к разработке надежного программного обеспечения.

ЗАЧЕМ ПИСАТЬ БЕЗОПАСНЫЙ КОД? (3)



Качественное ПО

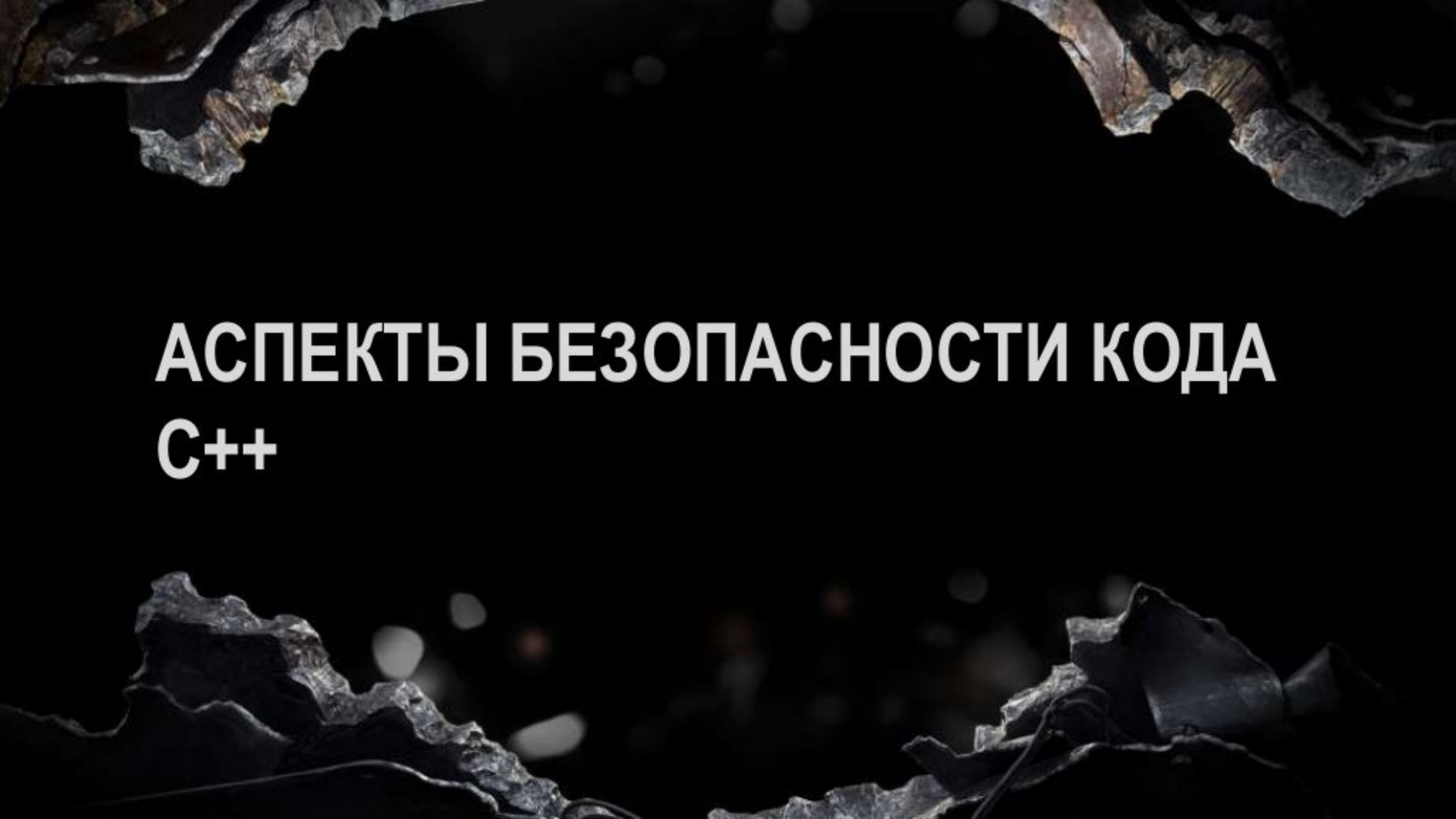
HAPPINESS IS



**...when your code
runs without error.**

for more visit www.dailyhappyquotes.com





АСПЕКТЫ БЕЗОПАСНОСТИ КОДА

C++

АСПЕКТЫ БЕЗОПАСНОСТИ КОДА

Безопасность границ памяти (bounds safety)

Безопасность времени жизни объектов (lifetime safety)

Безопасность типов (type safety)

Безопасность арифметических операций (arithmetic safety)

Безопасность относительно исключений (exception safety)

Безопасность работы в многопоточной среде (thread safety)

Защищенный код (security)



БЕЗОПАСНОСТЬ ГРАНИЦ ПАМЯТИ

БЕЗОПАСНОСТЬ ГРАНИЦ ПАМЯТИ (BOUNDS SAFETY)

- Чтение или запись данных за пределами выделенной области памяти позволяет очень скоро познакомиться с лучшим другом C++ разработчика — undefined behavior.



BUFFER OVERFLOW

- Запись в память за пределами выделенного участка (buffer overflow) может привести к
 - аварийному завершению приложения с ошибкой сегментации памяти (SIGSEGV, ACCESS_VIOLATION);
 - неожиданному поведению из-за модификации данных, не связанных с выполняемым в данный момент кодом;
 - порче пользовательских данных;
 - самому худшему — программа продолжит работать без обнаруживаемых проблем.

КАК РОЖДАЮТСЯ ОШИБКИ BUFFER OVERFLOW

- Использование адресной арифметики;
- Выход за пределы массива;
- Использование небезопасных (unchecked) функций работы с диапазонами данных;
- Нарушение правил безопасности типов (type safety).

ИСПОЛЬЗОВАНИЕ АДРЕСНОЙ АРИФМЕТИКИ

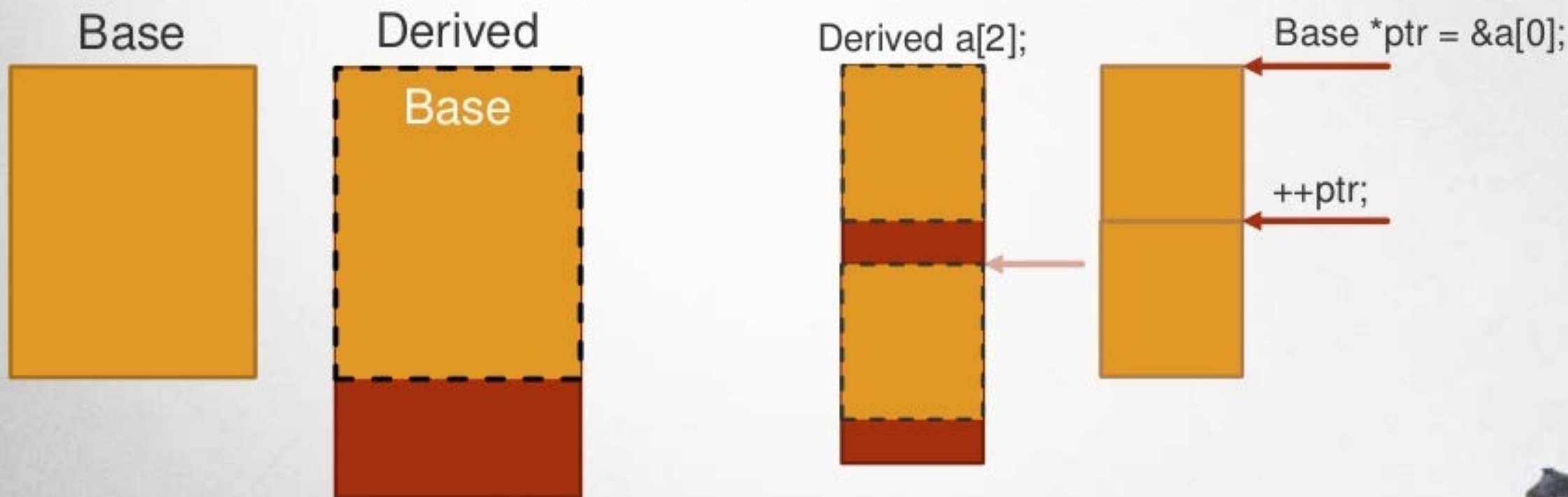
```
struct big{
    unsigned long long ull_1; /* typically 8 bytes */
    unsigned long long ull_2; /* typically 8 bytes */
    unsigned long long ull_3; /* typically 8 bytes */
    int si_4; /* typically 4 bytes */
    int si_5; /* typically 4 bytes */
};
/* ... */
size_t skip = offsetof(struct big, ull_2);
struct big *s = (struct big *)malloc(sizeof(struct big));
if (!s) {
    /* Handle malloc() error */
}

memset(s + skip, 0, sizeof(struct big) - skip);
/* ... */
free(s);
s = NULL;
```

```
memset((char *)s + skip, 0, sizeof(struct big) - skip);
```

ИСПОЛЬЗОВАНИЕ АДРЕСНОЙ АРИФМЕТИКИ (2)

- Не используйте адресную арифметику с полиморфным типами — это *undefined behavior*! Исключение составляют только классы со спецификатором `final`.



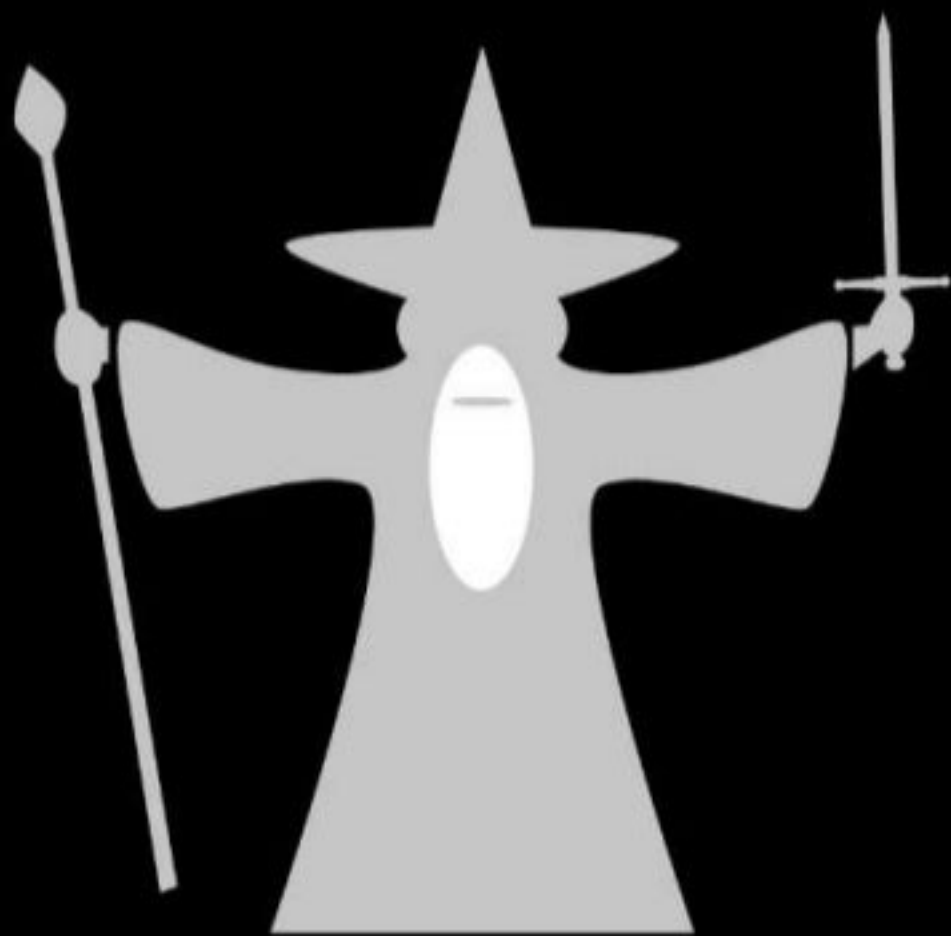
ИСПОЛЬЗОВАНИЕ АДРЕСНОЙ АРИФМЕТИКИ (3)

- › Не всегда интуитивно понятно;

```
T *ptr; int n;  
assert(reinterpret_cast<std::uintptr_t>(ptr + n) == reinterpret_cast<std::uintptr_t>(ptr) + n * sizeof(T));
```

- › Требуется ручной реализации контроля границ буфера;
- › Нельзя использовать с объектами полиморфных классов;
- › В целом, хрупкий и ошибкоопасный инструмент языка, требующий от разработчика предельной осторожности и внимания;
- › Использование контейнеров стандартной библиотеки или обертки над массивом `gs1::span` - гораздо лучший вариант.

YOU SHALL NOT PASS!



ВЫХОД ЗА ПРЕДЕЛЫ МАССИВА

- › оператор `[]` стандартных контейнеров C++ не проверяет границ, а функции `std::<container>::at()` — проверяют.

```
int values[10];  
values[0] = 1; // OK!  
values[-1] = 2; // Undefined behavior  
values[10] = 42; // Undefined behavior  
  
std::array<int, 10> values;  
values[0] = 1; // OK!  
values[11] = 2; // Undefined behavior  
values.at(11) = 5; // OK! Throws std::out_of_range  
values.at(-1) = 5; // OK! Throws std::out_of_range
```


ВЫХОД ЗА ПРЕДЕЛЫ МАССИВА (2)

- › Всегда следует проверять значение индекса (массива), который был получен извне (переменные окружения; аргументы функции main; пользовательский ввод; данные, прочитанные из файла или полученные по сети).

```
const size_t packet_types_count = 10;
const size_t packet_size[packet_types_count] = { /* ... */ };
/* ... */
int32 packet_type = read_packet_type(connection);
if (packet_type >= 0 && packet_type < packet_types_count)
{
    size_t size = packet_size[packet_type];
    /* ... */
}
else
{
    throw InvalidPacket();
}
```

ВЫХОД ЗА ПРЕДЕЛЫ МАССИВА (3)

- Аналогично, доступ к элементам динамических массивов по индексу требует проверки индекса относительно размера массива.

```
void main(int argc, char *argv[])
{
    static const int mode_arg_idx = 1;
    /*...*/
    if (argc > mode_arg_idx)
    {
        std::string mode_name = argv[mode_arg_idx];
        /*...*/
    }
    /*...*/
}
```

UNCHECKED ФУНКЦИИ СТАНДАРТНОЙ БИБЛИОТЕКИ

- › Использование небезопасных (unchecked) функций стандартной библиотеки повышает шансы возникновения ошибок переполнения буфера;
- › Включение в стандарт библиотеки Ranges и STL2 значительно упростит работу с диапазонами

<https://ericniebler.github.io/range-v3/>



UNCHECKED ФУНКЦИИ STL

```
std::vector<int> a, b, c;

// Unchecked output range
std::copy(std::begin(a), std::end(a), std::begin(b));
std::copy_n(std::begin(a), a.size(), std::begin(b));
std::copy_if(std::begin(a), std::end(a), std::begin(b),
    [](int value){ return value > 0; });
std::copy_backward(std::begin(a), std::end(a), std::end(b));
std::move(std::begin(a), std::end(a), std::begin(b));
std::move_backward(std::begin(a), std::end(a), std::end(b));
std::set_difference(std::begin(a), std::end(a), std::begin(b), std::end(b), std::begin(c));
std::set_symmetric_difference(std::begin(a), std::end(a), std::begin(b), std::end(b), std::begin(c));
std::set_intersection(std::begin(a), std::end(a), std::begin(b), std::end(b), std::begin(c));
std::set_union(std::begin(a), std::end(a), std::begin(b), std::end(b), std::begin(c));
std::merge(std::begin(a), std::end(a), std::begin(b), std::end(b), std::begin(c));
std::swap_ranges(std::begin(a), std::end(a), std::begin(b));

// Unchecked input range
std::mismatch(std::begin(a), std::end(a), std::begin(b));
std::equal(std::begin(a), std::end(a), std::begin(b));
std::is_permutation(std::begin(a), std::end(a), std::begin(b));

// Checked since C++17
std::mismatch(std::begin(a), std::end(a), std::begin(b), std::end(b));
std::equal(std::begin(a), std::end(a), std::begin(b), std::end(b));
std::is_permutation(std::begin(a), std::end(a), std::begin(b), std::end(b));
```

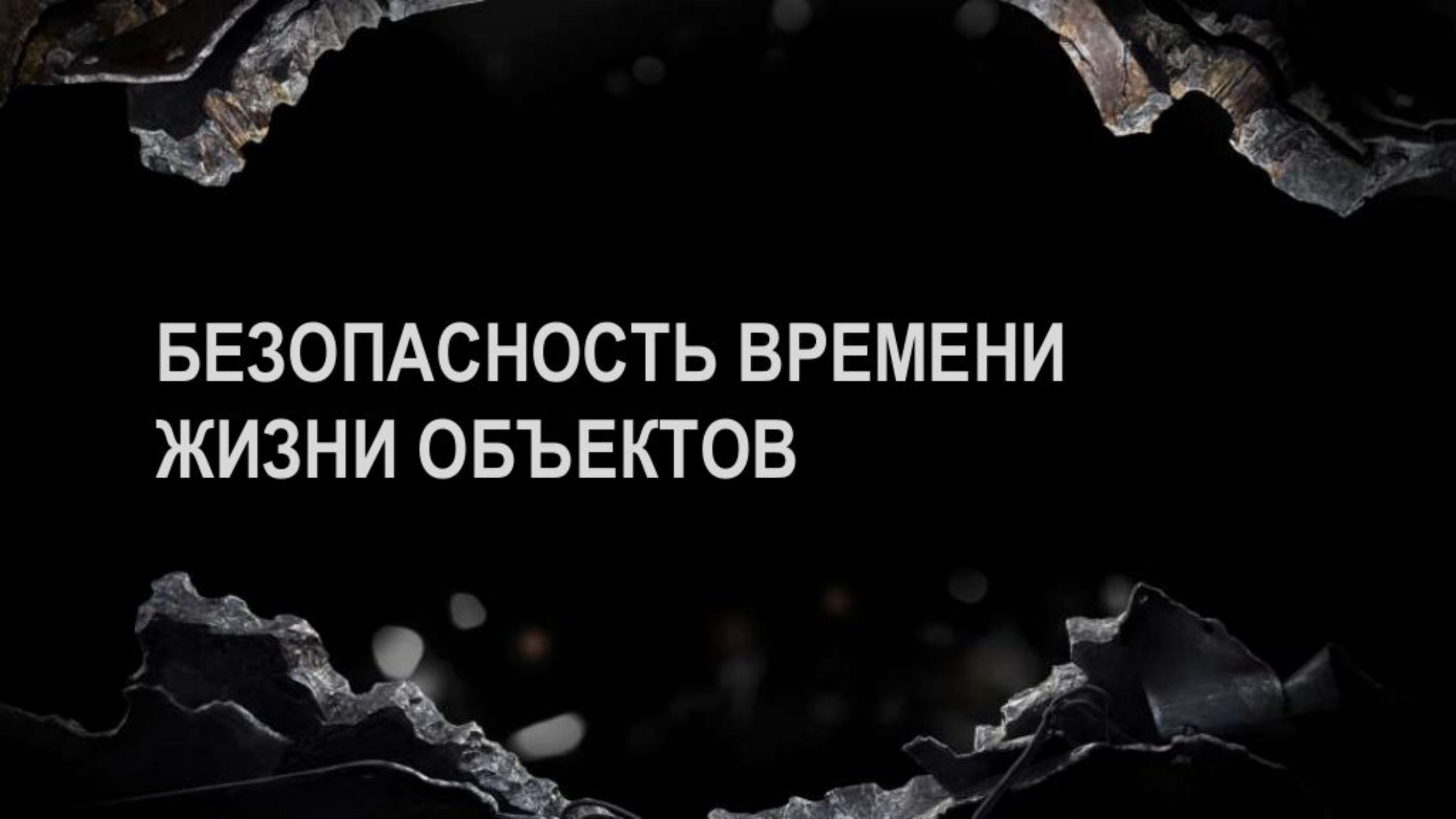
UNCHECKED ФУНКЦИИ STL (2)

Реализовать операции, выполняемые этими функциями всегда можно безопасным способом:

```
std::vector<int> a, b;  
/* ... */  
  
// Safe #1  
if (a.size() <= b.size())  
{  
    std::copy(std::begin(a), std::end(a), std::begin(b));  
}  
  
// Safe #2  
b.reserve(a.size());  
std::copy(std::begin(a), std::end(a), std::back_inserter(b));  
  
// Safe #3  
b.insert(std::end(b), std::begin(a), std::end(a));
```

UNCHECKED ФУНКЦИИ СТАНДАРТНОЙ БИБЛИОТЕКИ C

Функция	Аналог из STL
<code>std::memset</code>	<code>std::fill</code> , <code>std::array<>::fill</code> , <code>{}</code>
<code>std::strcpy</code> , <code>std::strncpy</code>	<code>std::string</code>
<code>std::strlen</code>	
<code>std::strcat</code> , <code>std::strncat</code>	
<code>std::memcpy</code> , <code>std::memmove</code> , <code>std::memcmp</code>	<code>std::array</code> , <code>std::vector</code>
<code>std::gets</code>	<code>std::fgets</code> , <code>std::getline</code>
<code>std::sprintf</code> , <code>std::vsprintf</code> , <code>std::vsnprintf</code> , ...	<code>iostream</code>
<code>std::scanf</code> , <code>std::vfscanf</code> , <code>std::vscanf</code> , ...	
<code>std::strtok</code>	<code>std::string</code> , <code>std::regex</code>



БЕЗОПАСНОСТЬ ВРЕМЕНИ ЖИЗНИ ОБЪЕКТОВ

JELLYFISH IN ARMOUR



CHRIS HENDRICKS © 2009

HOW MILK CONTAINERS SHOULD BE

БЕЗОПАСНОСТЬ ВРЕМЕНИ ЖИЗНИ (LIFETIME SAFETY)

- Чтобы обеспечить безопасность времени жизни объектов следует
 - Не использовать объекты до того, как они были успешно проинициализированы;
 - Удалять созданные в куче объекты;
 - Удалять созданные в куче объекты только единожды;
 - Не использовать объекты после истечения их времени жизни.

ИСПОЛЬЗУЙТЕ RAII

- › Всегда отдавайте предпочтение использованию идиомы RAII ручному управлению ресурсами.

```
// Manual resource management
Foo *bar = new Foo();
/*...*/
delete bar;

Foo *lots_ofBars = new Foo[100];
/*...*/
delete []lots_ofBars;

// RAII way
std::unique_ptr<Foo> bar = std::make_unique<Foo>();
std::vector<Foo> lots_ofBars(100);
```

НЕ СОЗДАВАЙТЕ ВИСЯЧИЕ ССЫЛКИ

- › Не выносите ссылки на локальные/временные объекты за пределы их области видимости.

```
// returned reference is dangling
const std::string& get_value()
{
    std::string value;
    /* ... */
    return value;
}

// str is dangling
const char *str;
{
    std::string value;
    /* ... */
    str = value.c_str();
}
```

```
// captured reference is dangling
std::function<void()> f;
{
    std::string value;
    /* ... */
    f = [&]() {
        std::cout << value;
    };
}
```

НЕ СОЗДАВАЙТЕ ВИСЯЧИЕ ССЫЛКИ (2)

- › Будьте осторожны с функциями, которые возвращают ссылки или прокси-объекты.

```
// r, max_str, min_str are dangling
const std::string& max_str = std::max("hello"s, "world"s);
const std::string& min_str = std::min("hello"s, "world"s);
const int& r = std::clamp(-1, 0, 255);

// beware of std::vector<bool>
std::vector<bool> some_bools{ /* ... */ };
auto first_bool = some_bools[0];

some_bools.clear();
// first_bool is invalid
```


УЧИТЫВАЙТЕ ВОЗМОЖНОСТЬ ИНВАЛИДАЦИИ ИТЕРАТОРОВ

- › Модификация контейнеров может привести к инвалидации итераторов, которые были созданы ранее. Невалидный итератор — это также разновидность висячей ссылки.

```
std::vector<int> numbers{ 0, 1, 2, 3, 4 };  
auto number_it = numbers.begin();  
  
numbers.push_back(5); // probably involves memory reallocation  
//number_it may be invalid
```

The background is a deep black field filled with numerous small, out-of-focus white and grey circular particles, creating a bokeh effect. Scattered across the top and bottom edges are fragments of dark, metallic-looking material, possibly broken metal or stone, with some internal textures visible.

БЕЗОПАСНОСТЬ ТИПОВ



БЕЗОПАСНОСТЬ ТИПОВ (TYPE SAFETY)

- › Не использовать `reinterpret_cast` (в том числе через `union`);
- › Отказаться от `static_cast` для приведения вниз по иерархии (`downcast`);
- › Не использовать понапрасну `const_cast`;
- › Не использовать приведение типов в стиле C;
- › Вместо `union` использовать `variant` (например, `boost::variant`);
- › Вместо функций с переменным числом аргументов использовать (`varargs`) использовать `variadic templates`;
- › Соблюдать One Definition Rule.

ИСПОЛЬЗОВАНИЕ REINTERPRET_CAST

- › Оператор позволяет убедить компилятор интерпретировать биты объекта одного типа как объект другого типа, никак с первым типом не связанного;
- › В общем случае гарантии корректности такого преобразования отсутствуют;
- › Подталкивает к тому, чтобы делать предположения о способе представления данных в памяти, что само по себе ненадежно и ведет к написанию некроссплатформенного кода;
- › Нельзя использовать для приведения полиморфных типов, поскольку `reinterpret_cast` никак не учитывает особенности размещения таких объектов;



ИСПОЛЬЗОВАНИЕ CONST_CAST

- Позволяет избавиться от спецификаторов `const` и `volatile`;
- Добавляет неожиданные для разработчика side-эффекты;
- В случае если полученная после преобразования `non-const` ссылка связана с настоящей константой, ее модификация приведет к `undefined behavior`;
- Компилятор неявно выполняет `const_cast` для строковых литералов (к `char*`), так как это необходимо для работы с функциями стандартной библиотеки C. Но это не значит, что строковые литералы можно модифицировать!;
- Использование оправдано только при работе с библиотеками C и API, которые не используют спецификатор `const` при отсутствии side-эффектов.

ИСПОЛЬЗОВАНИЕ C-STYLE CAST

- Производит одно из следующих преобразований:
 - `const_cast`
 - `static_cast`
 - `static_cast + const_cast`
 - `reinterpret_cast`
 - `reinterpret_cast + const_cast`
- Нет возможности узнать, какое именно преобразование произошло или как-то повлиять на этот выбор;
- Поведение может быть разным при одних и тех же преобразованиях в зависимости от наличия/отсутствия определения преобразуемых типов.

СОБЛЮДАЙТЕ ONE DEFINITION RULE

- Нарушение ODR - undefined behavior. Ошибки, связанные с нарушением ODR сложно диагностировать.

Foo.h

```
#pragma once

class Foo
{
    /* ... */
private:
    int a1;
#ifdef ENABLE_SUPER_FEATURE
    int a2;
    int a3;
#endif
};
```

GetFooSize1.cpp

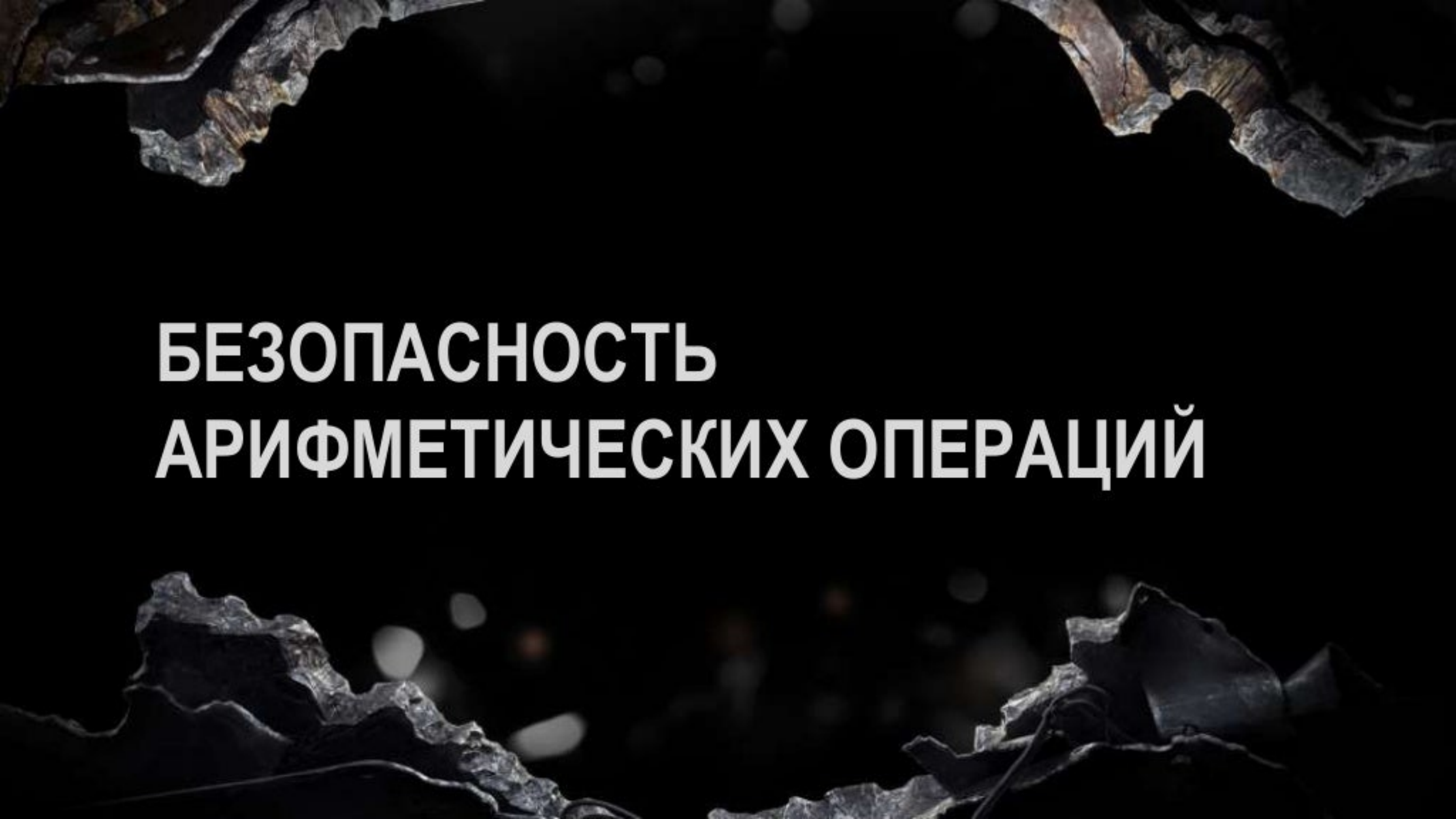
```
#define ENABLE_SUPER_FEATURE
/* ... */
#include "Foo.h"

size_t GetFooSize1()
{
    return sizeof(Foo);
}
// returns 12
```

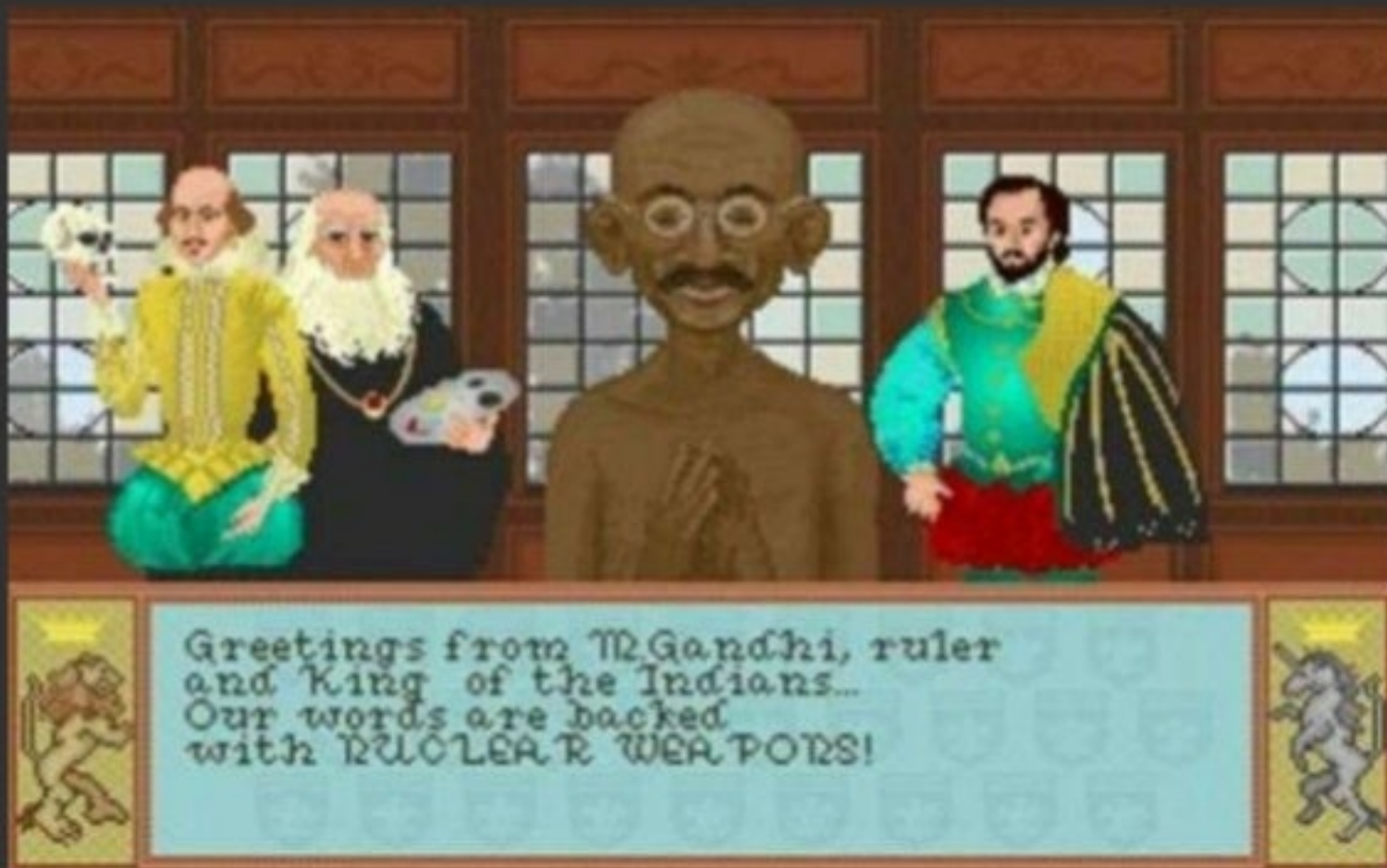
GetFooSize2.cpp

```
/* ... */
#include "Foo.h"

size_t GetFooSize2()
{
    return sizeof(Foo);
}
// returns 4
```



БЕЗОПАСНОСТЬ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ



Greetings from Mr Gandhi, ruler
and King of the Indians...
Our words are backed
with NUCLEAR WEAPONS!

БЕЗОПАСНОСТЬ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ

*0 программистов ругал сердитый шеф,
Потом уволил одного, и стало их FF.*

(с) Компьютерра

- Необходимо учитывать возможность «unsigned integer wrapping» при совершении арифметических операций;
- Перед выполнением арифметических операций с целыми числами стоит удостовериться в том, что не произойдет переполнение. Переполнение знакового целого – undefined behavior;
- Результат операции целочисленного деления на 0 (или взятие остатка от деления на 0) не определен;

БЕЗОПАСНОСТЬ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ (2)

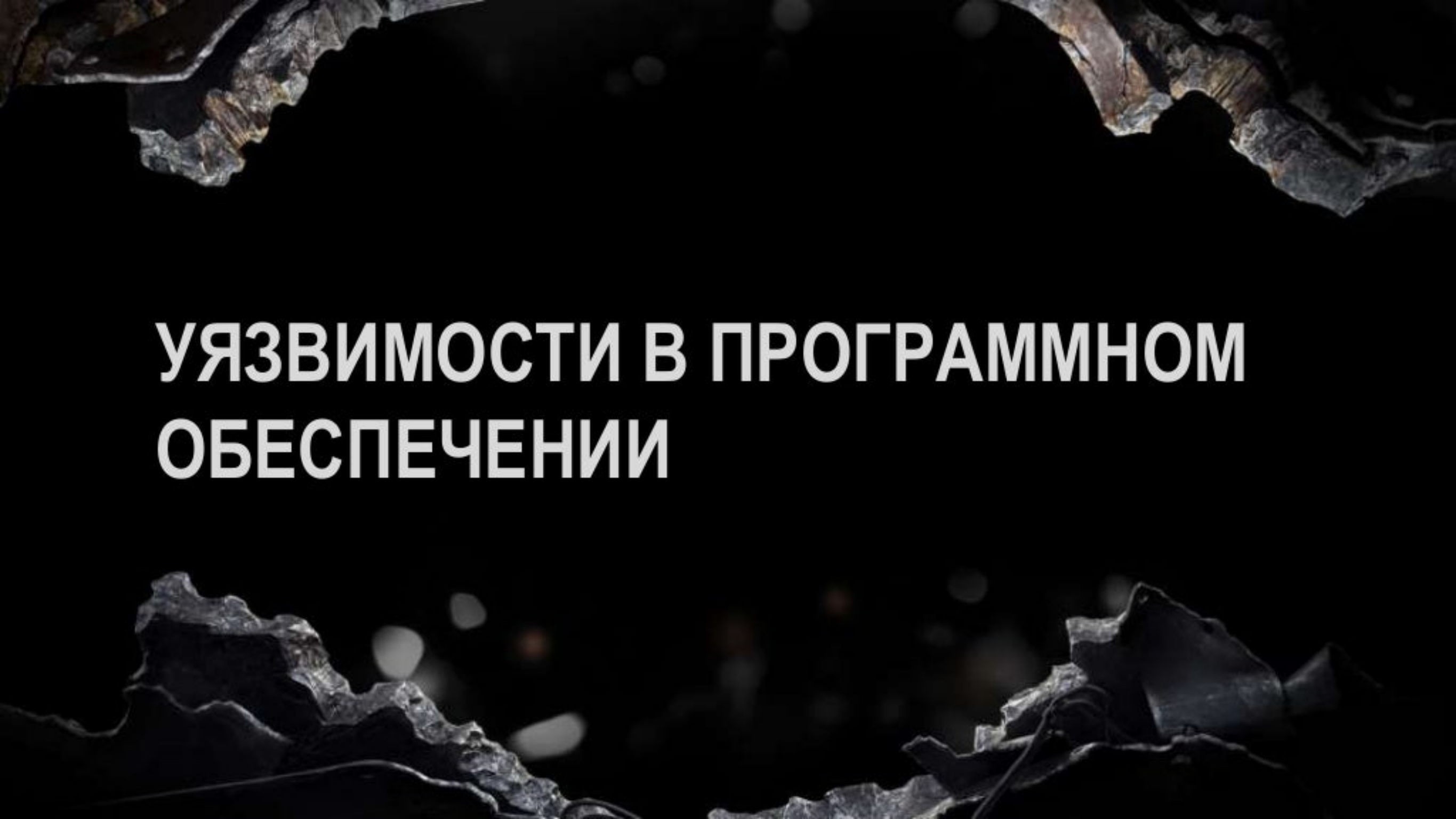
```
int a = std::numeric_limits<int>::max();
int b;

b = a + 1;    // Undefined behavior
b = -a;       // OK!
/* int: -2,147,483,648 .. 2,147,483,647 */
b = a / 0;    // Undefined behavior
b = a % 0;    // Undefined behavior

a = std::numeric_limits<int>::min();
b = a - 1;    // Undefined behavior
b = -a;       // Undefined behavior
unsigned int c = std::numeric_limits<unsigned int>::max();

++c;          // OK! Wrapping around
assert(c == std::numeric_limits<unsigned int>::min());
assert(c == 0);

--c;          // OK! Wrapping around
assert(c == std::numeric_limits<unsigned int>::max());
```

УЯЗВИМОСТИ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ



CWE/SANS TOP 25 MOST DANGEROUS SOFTWARE ERRORS

Top 25 Most Dangerous Software Errors – результат работы экспертов в области безопасности ПО из SANS Institute и MITRE.

Rank	Name
[3]	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[18]	Use of Potentially Dangerous Function
[20]	Incorrect Calculation of Buffer Size
[23]	Uncontrolled Format String
[24]	Integer Overflow or Wraparound

ЧТО ПОЧИТАТЬ?

- › C++ Core Guidelines
<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- › SEI CERT C++ Coding Standard
<https://www.securecoding.cert.org/>
- › OWASP C/C++ Technology Initiative
<https://www.owasp.org/index.php/C%2B%2B>
- › Common Weakness Enumeration by MITRE Corporation
<http://cwe.mitre.org/>

ВОПРОСЫ?

ЛАПИЦКИЙ АРТЕМ

Software Developer



lapitsky.artem@gmail.com



<https://www.facebook.com/WargamingMinsk>



<https://www.linkedin.com/company/wargaming-net>

wargaming.com



СПАСИБО ЗА ВНИМАНИЕ!



WARGAMING.NET
LET'S BATTLE