

# UNDEFINED BEHAVIOR: КАК ПРОСТРЕЛИТЬ СЕБЕ НОГУ В C++

Петр Калинин, 2016

Эту презентацию можно распространять по лицензии  
GNU General Public License версии 3.0 или старше



# Компиляторы C++

---

# Компиляторы C++

---

- Gnu Compiler Collection (GCC)

# Компиляторы C++

---

- Gnu Compiler Collection (GCC)
- Microsoft Visual Studio (MSVS)

# Компиляторы C++

---

- Gnu Compiler Collection (GCC)
- Microsoft Visual Studio (MSVS)
- Clang

# Компиляторы C++

---

- Gnu Compiler Collection (GCC)
- Microsoft Visual Studio (MSVS)
- Clang
- Intel C Compiler (ICC)
- Много много других

# Стандарт C++

---

- Публикуется International Standard Organisation
- Большинство компиляторов стараются его соблюдать (MSVS известное исключение)
- Существуют разные версии стандарта: C++03, C++11 (C++0x), C++14, C++17 (в разработке)
- Отдельные аспекты языка отданы «на откуп» авторам компиляторов («реализаций»)

# Вариации стандарта

---

- Implementation-defined behavior
  - Реализация (компилятор) должен выбрать вариант, документировать и придерживаться его
  - Пример: размер типа `int`



# Вариации стандарта

---

- Implementation-defined behavior
  - Реализация (компилятор) должен выбрать вариант, документировать и придерживаться его
  - Пример: размер типа `int`
- Unspecified behavior
  - Компилятор может выбрать любой из вариантов, причем каждый раз новый
  - Пример: порядок вычислений аргументов функции

# Пример Unspecified behavior

---

```
#include <iostream>

void foo(int a, int b) {
}

int bar() {
    std::cout << "bar()" << std::endl;
    return 10;
}

int buz() {
    std::cout << "buz()" << std::endl;
    return 20;
}

int main() {
    foo(bar(), buz());
    return 0;
}
```

# Пример Unspecified behavior

```
#include <iostream>

void foo(int a, int b) {

}

int bar() {
    std::cout << "bar()" << std::endl;
    return 10;
}

int buz() {
    std::cout << "buz()" << std::endl;
    return 20;
}

int main() {
    foo(bar(), buz());
    return 0;
}
```

```
$ g++ evaluation_order.cpp \
    -o evaluation_order
$ ./evaluation_order
buz()
bar()

$ clang++ evaluation_order.cpp \
    -o evaluation_order
$ ./evaluation_order
bar()
buz()
```

# Вариации стандарта

---

- Implementation-defined behavior
  - Реализация (компилятор) должен выбрать вариант, документировать и придерживаться его
  - Пример: размер типа `int`
- Unspecified behavior
  - Компилятор может выбрать любой из вариантов, причем каждый раз новый
  - Пример: порядок вычислений аргументов функции

# Вариации стандарта

- Implementation-defined behavior
  - Реализация (компилятор) должен выбрать вариант, документировать и придерживаться его
  - Пример: размер типа `int`
- Unspecified behavior
  - Компилятор может выбрать любой из вариантов, причем каждый раз новый
  - Пример: порядок вычислений аргументов функции
- Undefined behavior
  - Компилятор (и скомпилированная программа) может сделать абсолютно что угодно
  - Диагностировать ошибку компиляции, «вылететь», выдать некорректные данные, отформатировать жесткий диск...
  - А может и работать «как ожидалось»

# Undefined behavior подробнее

- Стандарт позволяет компилятору сделать что угодно вообще
- Никаких требований к поведению компилятора/программы нет
- Программа может скомпилироваться, не скомпилироваться с ошибкой, компилятор может упасть и т.д.
- Скомпилированная программа может работать как ожидалось, молча работать неправильно, падать, форматировать жесткий диск и т.д.
- «Nasal daemos»: «...can make daemons fly out of your nose»
- В частности, программа может «предвидеть»: ошибка может произойти еще до того, как программа дошла до ошибочного места
- На самом деле, UB используется в первую очередь для оптимизации и переносимости
- В C ситуация аналогичная

# Выход за пределы массива

---

```
#include <iostream>

int table[4] = {2, 4, 6, 8};

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

int main() {
    for (int i=0; i<10; i++) {
        std::cout << i << " " << exists_in_table(i) << std::endl;
    }
    return 0;
}
```

# Выход за пределы массива

---

```
$ g++ array_range_error.cpp \  
    -o array_range_error
```

```
$ ./array_range_error
```

```
0 1
```

```
1 0
```

```
2 1
```

```
3 0
```

```
4 1
```

```
5 0
```

```
6 1
```

```
7 0
```

```
8 1
```

```
9 0
```



# Выход за пределы массива

```
$ g++ array_range_error.cpp \  
-o array_range_error
```

```
$ ./array_range_error
```

```
0 1  
1 0  
2 1  
3 0  
4 1  
5 0  
6 1  
7 0  
8 1  
9 0
```

```
$ g++ -O3 array_range_error.cpp \  
-o array_range_error
```

```
$ ./array_range_error
```

```
0 1  
1 1  
2 1  
3 1  
4 1  
5 1  
6 1  
7 1  
8 1  
9 1
```

# Неинициализированный bool

```
#include <iostream>

void foo() {
    bool b;
    if (b)
        std::cout
            << "b is true"
            << std::endl;
    if (!b)
        std::cout
            << "b is not true"
            << std::endl;
}

void bar() {
    char a = 12;
}

int main() {
    bar();
    foo();
    return 0;
}
```

# Неинициализированный bool

```
#include <iostream>

void foo() {
    bool b;
    if (b)
        std::cout
            << "b is true"
            << std::endl;
    if (!b)
        std::cout
            << "b is not true"
            << std::endl;
}

void bar() {
    char a = 12;
}

int main() {
    bar();
    foo();
    return 0;
}
```

```
$ g++ uninitialized_bool.cpp \
    -o uninitialized_bool
```

```
$ ./uninitialized_bool
b is true
b is not true
```

# Переполнение int'a

```
#include <iostream>
#include <climits>

int foo ( int x) {
    return ( x+1 ) > x ;
}

int main ( void ) {
    std::cout
        << ( (INT_MAX+1) > INT_MAX )
        << std::endl;
    std::cout
        << foo ( INT_MAX )
        << std::endl;
    return 0;
}
```

# Переполнение int'a

```
#include <iostream>
#include <climits>

int foo ( int x) {
    return ( x+1 ) > x ;
}

int main ( void ) {
    std::cout
        << ( (INT_MAX+1) > INT_MAX )
        << std::endl;
    std::cout
        << foo ( INT_MAX )
        << std::endl;
    return 0;
}
```

```
$ g++ signed_overflow.cpp -o signed_ov
signed_overflow.cpp:
In function 'int main()':
signed_overflow.cpp:9:28:
warning: integer overflow in expression
        std::cout << ( (INT_MAX+1) > INT_

$ ./signed_overflow
0
0

$ g++ -O3 signed_overflow.cpp -o signe
signed_overflow.cpp:
In function 'int main()':
signed_overflow.cpp:9:28:
warning: integer overflow in expression
        std::cout << ( (INT_MAX+1) > INT_
                                ^

$ ./signed_overflow
0
1
```

# Изменение const-переменной

```
#include <iostream>

void f(const int& a) {
    const_cast<int&>(a) = 10;
    std::cout << "a=" << a
                << std::endl;
}

const int a = 5;

int main() {
    std::cout << "a=" << a
                << std::endl;
    f(a);
    std::cout << "a=" << a
                << std::endl;
    return 0;
}
```

# Изменение const-переменной

```
#include <iostream>

void f(const int& a) {
    const_cast<int&>(a) = 10;
    std::cout << "a=" << a
               << std::endl;
}

const int a = 5;

int main() {
    std::cout << "a=" << a
               << std::endl;
    f(a);
    std::cout << "a=" << a
               << std::endl;
    return 0;
}
```

```
$ g++ modify_const.cpp \
    -o modify_const

$ ./modify_const
a=5
Ошибка сегментирования
(сделан дамп памяти)
```

# Изменение const-переменной

```
#include <iostream>

void f(const int& a) {
    std::cout << "f:a=" << a
               << std::endl;
    const_cast<int&>(a) = 10;
    std::cout << "f:a=" << a
               << std::endl;
}

int main() {
    const int a = 5;
    std::cout << "main:a=" << a
               << std::endl;
    f(a);
    std::cout << "main:a=" << a
               << std::endl;
    f(a);
    std::cout << "main:a=" << a
               << std::endl;
    return 0;
}
```



# Изменение const-переменной

```
#include <iostream>
void f(const int& a) {
    std::cout << "f:a=" << a
               << std::endl;
    const_cast<int&>(a) = 10;
    std::cout << "f:a=" << a
               << std::endl;
}

int main() {
    const int a = 5;
    std::cout << "main:a=" << a
               << std::endl;
    f(a);
    std::cout << "main:a=" << a
               << std::endl;
    f(a);
    std::cout << "main:a=" << a
               << std::endl;
    return 0;
}
```

```
$ g++ modify_const_2.cpp \
    -o modify_const_2

$ ./modify_const_2
main:a=5
f:a=5
f:a=10
main:a=5
f:a=10
f:a=10
main:a=5
```

# Слишком большой сдвиг

---

```
#include <iostream>

int main() {
    int a = 1;
    int b = 32;
    std::cout << (a << b)
                << std::endl;
}
```

# Слишком большой сдвиг

```
#include <iostream>

int main() {
    int a = 1;
    int b = 32;
    std::cout << (a << b)
                << std::endl;
}
```

```
$ g++ left_shift.cpp -o left_shift
```

```
$ ./left_shift
```

```
1
```

```
$ g++ -O3 left_shift.cpp \
    -o left_shift
```

```
$ ./left_shift
```

```
0
```

# Двухкратная модификация

---

```
#include <iostream>

int main() {
    int i = 0;
    i = i++ + ++i;
    std::cout << i << std::endl;
    return 0;
}
```

# Двухкратная модификация

---

```
void f(int, int);
```

```
...
```

```
int i = 0;
```

```
f(i=-1, i=-1);
```

# Двухкратная модификация

---

<code>void f(int, int);</code>	<code>ZERO i</code>
	<code>DEC i</code>
<code>...</code>	<code>ZERO i</code>
	<code>DEC i</code>
<code>int i = 0;</code>	<code>CALL f</code>
<code>f(i=-1, i=-1);</code>	

# Двухкратная модификация

---

```
void f(int, int);
```

```
...
```

```
int i = 0;
```

```
f(i=-1, i=-1);
```

```
ZERO i
```

```
DEC i
```

```
ZERO i
```

```
DEC i
```

```
CALL f
```

```
ZERO i
```

```
ZERO i
```

```
DEC i
```

```
DEC i
```

```
CALL f
```

# realloc

---

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int*)malloc(sizeof(int));
    int *q =
        (int*)realloc(p, sizeof(int));
    *p = 1;
    *q = 2;
    if (p == q)
        printf("%d %d\n", *p, *q);
}
```



# realloc

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int*)malloc(sizeof(int));
    int *q =
        (int*)realloc(p, sizeof(int));
    *p = 1;
    *q = 2;
    if (p == q)
        printf("%d %d\n", *p, *q);
}
```

```
$ clang++ realloc.cpp -o realloc
```

```
$ ./realloc
```

```
2 2
```

```
$ clang++ -O3 realloc.cpp -o realloc
```

```
$ ./realloc
```

```
1 2
```

# Выход за пределы массива

---