Niesforne bity i bajty

Kilka słów

O mnie

@ senghe@gmail.com

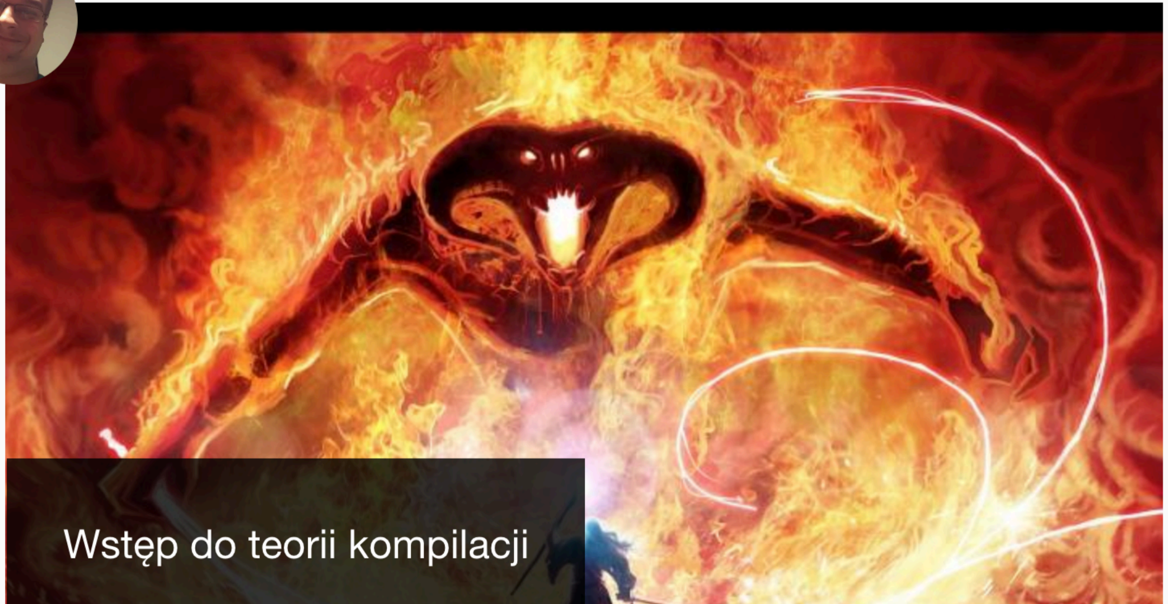
in https://www.linkedin.com/in/marcin-kukliński

cpp-polska.pl

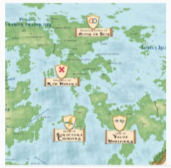
BLOG PROGRAMISTYCZNY

Wstęp do teorii kompilacji

Marcin Kukliński   kompilacja   *2018-11-28, 23:14*



Ostatnio na blogu



CppNews #51 [31.12.2018 - 06.01.2019]
2019-01-07

Jak używać

Ekipa cpp-polska

Marcin Kukliński

Kompilacja & Mario::Edit

Wojciech Razik

CppNews & Modern C++

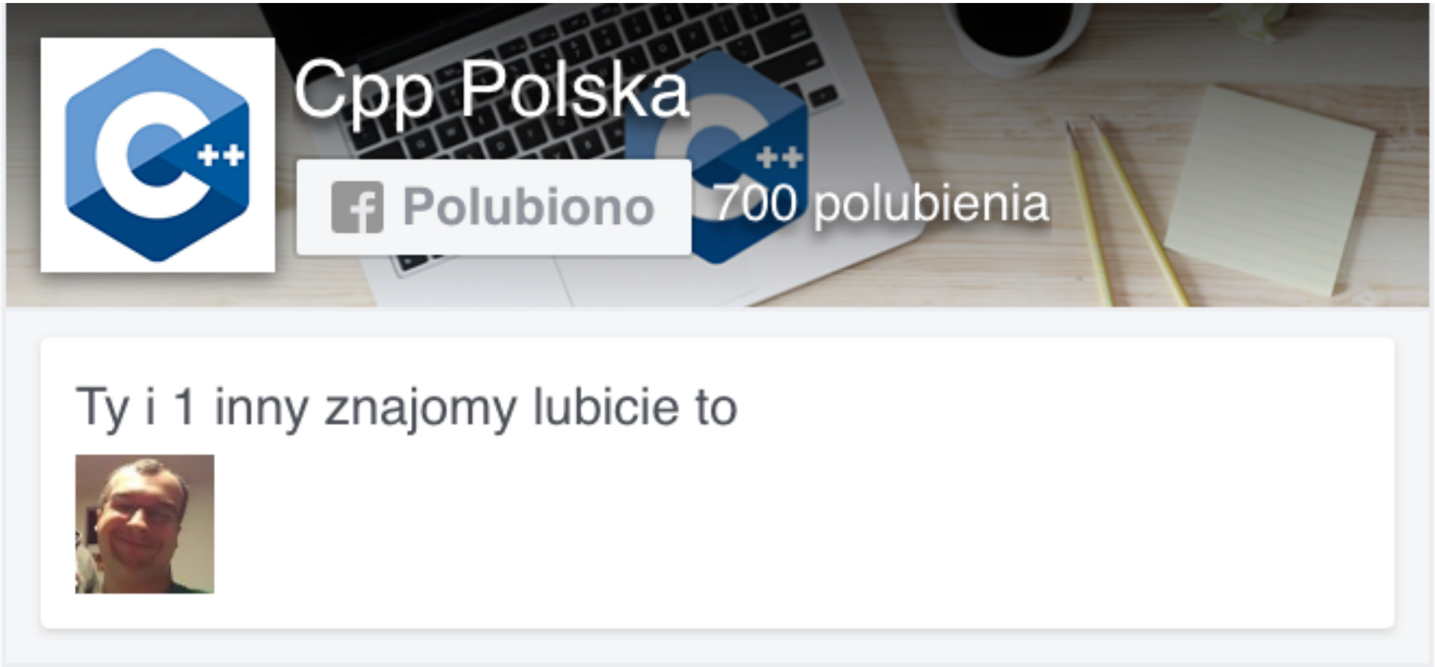
Michał Rudowicz

Modern C++

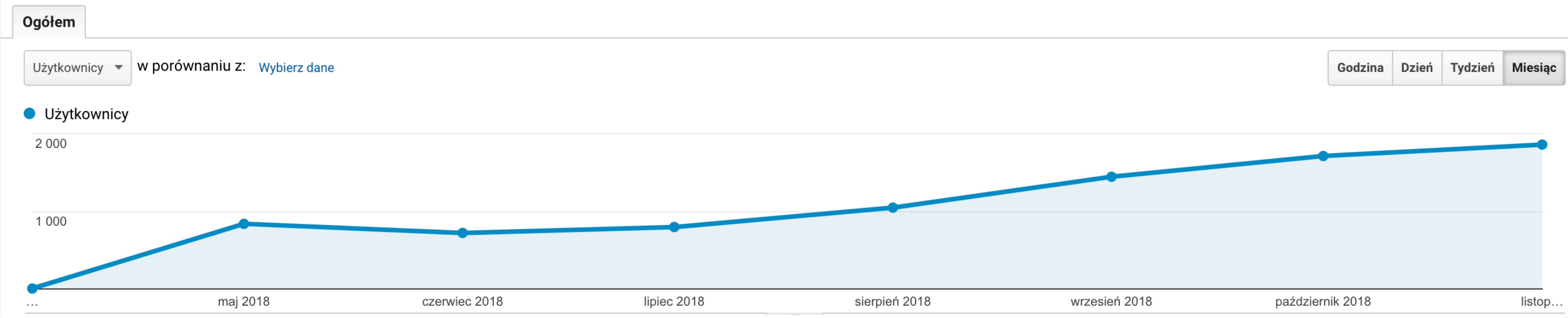
Bartłomiej Filipek

C++17

Nasze statystyki



Nasze statystyki



Znajdziecie nas na:

@CppPolskaOfficial

@CppPolska

@CppPolska

Naszym celem jest:

Propagowanie wiedzy o nowoczesnym C++

Może do nas dołączysz?

...

Ty

...

Wracając do bitów…

To jest bit

1

To też jest bit

0

A to jest bajt

11010111

Liczby całkowite

Kod uzupełnień do jedności (U1)

Kod uzupełnień do jedności (U1)

Kod uzupełnień do jedności to sposób zapisu liczb całkowitych oznaczany jako **ZU1** lub **U1**. Liczby dodatnie zapisywane są jak w naturalnym kodzie binarnym, przy czym najbardziej znaczący bit – traktowany jako bit znaku – musi mieć wartość 0. Do reprezentowania liczb ujemnych wykorzystywana jest bitowa negacja danej liczby, co sprawia, że bit znaku ma wartość 1.

Źródło: https://pl.wikipedia.org/wiki/Kod_uzupełnień_do_jedności

Kod uzupełnień do jedności (U1)

01000011

Kod uzupełnień do jedności (U1)

-2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

0 1 0 0 0 0 1 1

Kod uzupełnień do jedności (U1)

2^6 2^1 2^0

01000011

Kod uzupełnień do jedności (U1)

64 2 1

01000011

Kod uzupełnień do jedności (U1)

64 2 1

01000011 = 67

Kod uzupełnień do jedności (U1)

01000011 = 67

10111100

Kod uzupełnień do jedności (U1)

$01000011 = 67$

$-2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

10111100

Kod uzupełnień do jedności (U1)

01000011 = 67

-2^7 2^5 2^4 2^3 2^2

10111100

Kod uzupełnień do jedności (U1)

01000011 = 67

-128 32 16 8 4

10111100

Kod uzupełnień do jedności (U1)

01000011 = 67

-128 32 16 8 4 +1

10111100

Kod uzupełnień do jedności (U1)

01000011 = 67

-128 32 16 8 4 +1

10111100 = -67

Co z zerem?

U1 daje nam dwie wartości zerowe

Zero dodatnie:	00000000
Zero ujemne:	11111111

Kod uzupełnień do dwóch (U2)

Kod uzupełnień do dwóch (U2)

Kod uzupełnień do dwóch (w skrócie **U2** lub **ZU2**) – system reprezentacji liczb całkowitych w dwójkowym systemie pozycyjnym. Jest obecnie najpopularniejszym sposobem zapisu liczb całkowitych w systemach cyfrowych. (…) Zaletą tego kodu jest również istnienie tylko jednego zera. Przedział kodowanych liczb nie jest przez to symetryczny.

Źródło: https://pl.wikipedia.org/wiki/Kod_uzupełnień_do_dwóch

Kod uzupełnień do dwóch (U2)

01000011

Kod uzupełnień do dwóch (U2)

$-2^7 \; 2^6 \; 2^5 \; 2^4 \; 2^3 \; 2^2 \; 2^1 \; 2^0$

01000011

Kod uzupełnień do dwóch (U2)

2^6 $\qquad\qquad\qquad$ 2^1 2^0

01000011

Kod uzupełnień do dwóch (U2)

64 2 1

01000011

Kod uzupełnień do dwóch (U2)

64 2 1

01000011 = 67

Kod uzupełnień do dwóch (U2)

01000011 = 67

10111100

Kod uzupełnień do dwóch (U2)

01000011 = 67

10111101

Kod uzupełnień do dwóch (U2)

01000011 = 67

-2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

10111101

Kod uzupełnień do dwóch (U2)

01000011 = 67

-2^7 2^5 2^4 2^3 2^2 2^0

10111101

Kod uzupełnień do dwóch (U2)

01000011 = 67

-128   32 16 8 4   1

10111101

Kod uzupełnień do dwóch (U2)

01000011 = 67

-128 32 16 8 4 1

10111101 = -67

Kod uzupełnień do dwóch (U2)

unsigned char 0 .. 255

Kod uzupełnień do dwóch (U2)

unsigned char	0 .. 255
signed char	-128 .. 127

Dlaczego to jest ważne?

Co mówi standard...

"The range of representable values for a signed integer type is -2^{N-1} to $2^{N-1}-1$ (inclusive), where N is called the *range exponent* of the type"

Źródło: http://eel.is/c++draft/basic.fundamental

Integer Overflow

Integer Overflow

Wyrzucenie wyjątku

```
INT_MAX + 10 = Exception
```

Integer Overflow

Przycięcie wyniku

```
INT_MAX + 10 = 10
```

Integer Overflow

Saturacja

```
INT_MAX + 10 = INT_MAX
```

Co mówi standard...

"For each of the standard signed integer types, there exists a corresponding (but different) standard unsigned integer type: "unsigned char", "unsigned short int", "unsigned int", "unsigned long int", and "unsigned long long int". Likewise, for each of the extended signed integer types, there exists a corresponding extended unsigned integer type. The standard and extended unsigned integer types are collectively called unsigned integer types. An unsigned integer type has the same range exponent N as the corresponding signed integer type.

The range of representable values for the unsigned type is 0 to 2^{N-1} (inclusive); arithmetic for the unsigned type is performed modulo 2^N. [Note: Unsigned arithmetic does not overflow. Overflow for signed arithmetic yields undefined behavior"

Źródło: http://eel.is/c++draft/basic.fundamental

Integer Overflow to UB

Undefined Behavior

- Nie mamy pewności, co zostanie wykonane przez procesor

Undefined Behavior

- Nie mamy pewności, co zostanie wykonane przez procesor
- Kompilator zakłada, że niezdefiniowane zachowanie nigdy nie ma miejsca

Undefined Behavior

- Nie mamy pewności, co zostanie wykonane przez procesor
- Kompilator zakłada, że niezdefiniowane zachowanie nigdy nie ma miejsca
- Cała odpowiedzialność zostaje zrzucona na programistę

UB vs Implementation Defined

Jak radzić sobie z Integer Overflow?


```
int add(int a, int b) {
    int sum = a + b;
    if (a > 0 && b > 0) {
        if (sum < a) {
            throw E();
        }
        if (sum < b) {
            throw E();
        }
    }
    return sum;
}
```

Jak radzić sobie z Integer Overflow?

```
int add(int a, int b) {
    int sum = a + b;
    if (a > 0 && b > 0) {
//      if (sum < a) {
//          throw E();
//      }
//      if (sum < b) {
//          throw E();
//      }
    }
    return sum;
}
```

Jak radzić sobie z Integer Overflow?


```
unsigned add(unsigned a, unsigned b) {
   if (INT_MAX-a < b) {
       throw E();
   }
   return a + b;
}
```

Jak radzić sobie z Integer Overflow?


```
unsigned add(unsigned a, unsigned b) {
    if (INT_MAX-a < b) {
        throw E();
    }
    return a + b;
}
```

Jak radzić sobie z Integer Overflow?


```
int add(int a, int b) {
    if (b > 0 && a > 0) {
        if (INT_MAX-a < b) {
            throw E();
        }
    } else if (b < 0 && a < 0) {
        if (INT_MIN-a > b) {
            throw E();
        }
    }
    return a + b;
}
```

Jak radzić sobie z Integer Overflow?


```
int add(int a, int b) {
    if (b > 0 && a > 0) {
        if (INT_MAX-a < b) {
            throw E();
        }
    } else if (b < 0 && a < 0) {
        if (INT_MIN-a > b) {
            throw E();
        }
    }
    return a + b;
}
```

Jak radzić sobie z Integer Overflow?


```
int add(int a, int b) {
    if (b > 0 && a > 0) {
        if (INT_MAX-a < b) {
            throw E();
        }
    } else if (b < 0 && a < 0) {
        if (INT_MIN-a > b) {
            throw E();
        }
    }
    return a + b;
}
```

Jak radzić sobie z Integer Overflow?


```
int add(int a, int b) {
    if (b > 0 && a > 0) {
        if (INT_MAX-a < b) {
            throw E();
        }
    } else if (b < 0 && a < 0) {
        if (INT_MIN-a > b) {
            throw E();
        }
    }
    return a + b;
}
```

Ukryty Integer Overflow

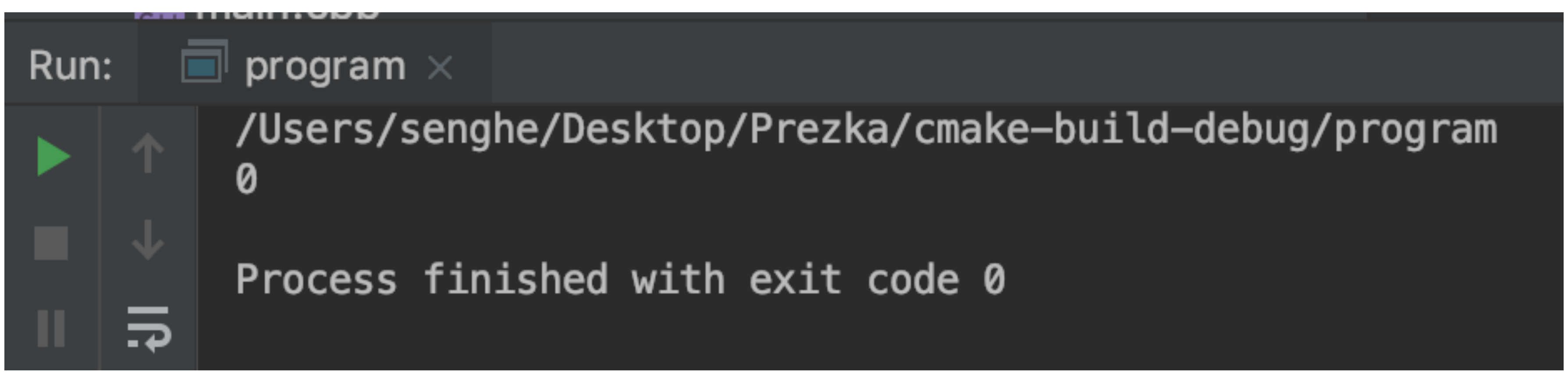
Ukryty Integer Overflow


```
int main(int argc, char* argv[]) {
    uint64_t v = 0x80000000 * 2;
    std::cout << std::hex << v << std::endl;
    return 0;
}
```

?

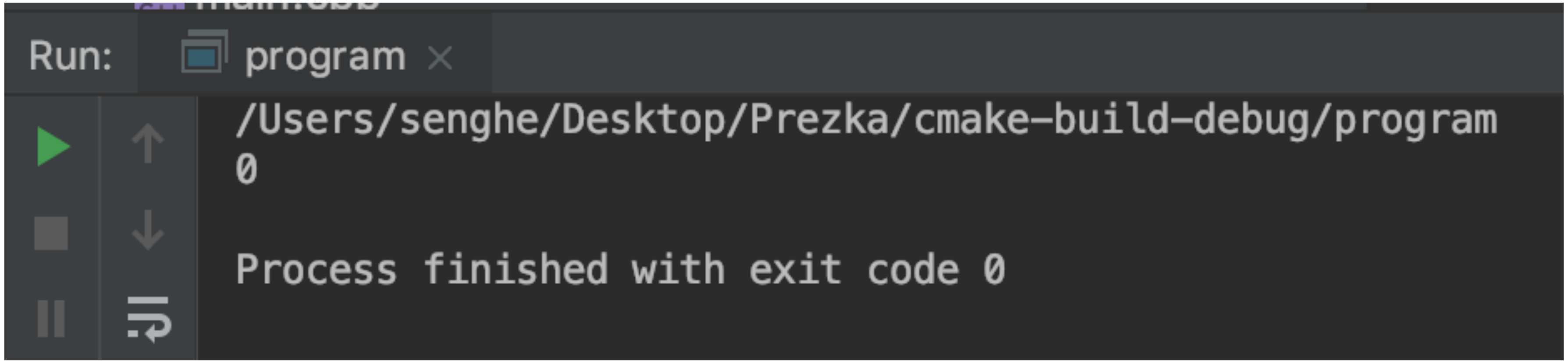
Ukryty Integer Overflow

```
int main(int argc, char* argv[]) {
    uint64_t v = 0x80000000 * 2;
    std::cout << std::hex << v << std::endl;
    return 0;
}
```



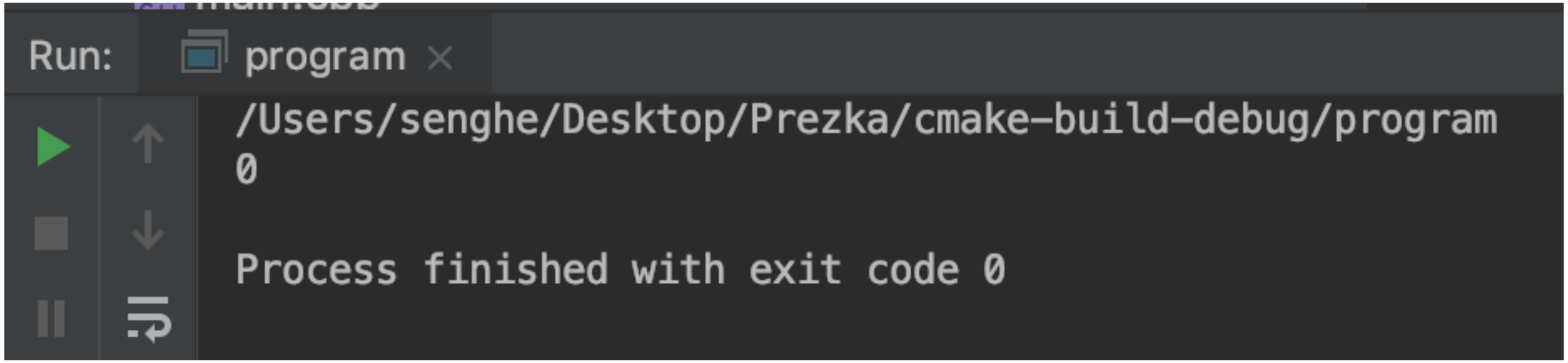
Ukryty Integer Overflow

```
int main(int argc, char* argv[]) {
    uint32_t v = 0x80000000 * 2;
    std::cout << std::hex << v << std::endl;
    return 0;
}
```



Ukryty Integer Overflow

```
int main(int argc, char* argv[]) {
    uint64_t v = 0x80000000 * 2;
    std::cout << std::hex << v << std::endl;
    return 0;
}
```



Ukryty Integer Overflow

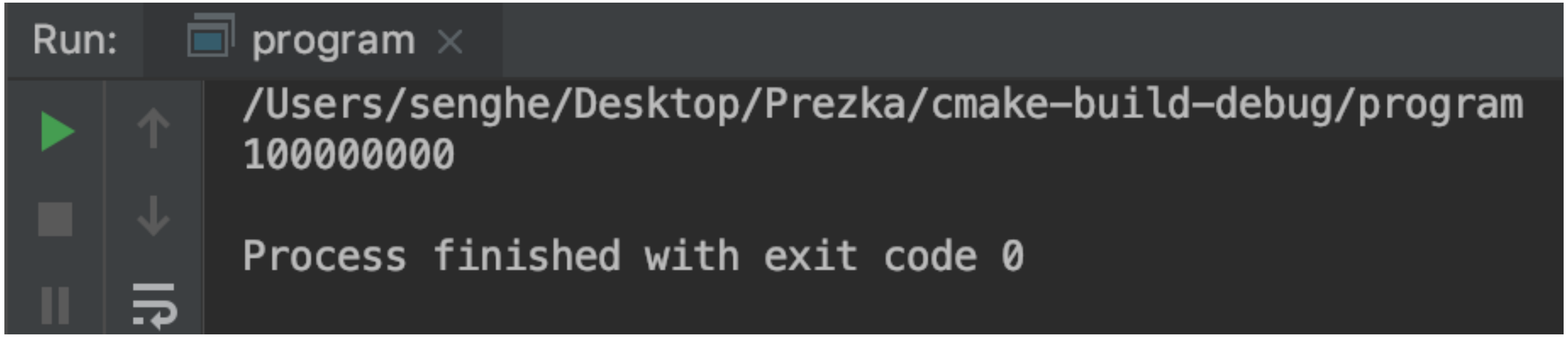

```
int main(int argc, char* argv[]) {
    uint64_t v = 0x80000000ULL * 2;
    std::cout << std::hex << v << std::endl;
    return 0;
}
```

Ukryty Integer Overflow


```
int main(int argc, char* argv[]) {
    uint64_t v = 0x80000000 * 2ULL;
    std::cout << std::hex << v << std::endl;
    return 0;
}
```

Ukryty Integer Overflow


```
int main(int argc, char* argv[]) {
    uint64_t v = 0x80000000 * 2ULL;
    std::cout << std::hex << v << std::endl;
    return 0;
}
```



Czym jest promocja typu?

Czym jest promocja typu

Promocją typu nazywamy proces polegający na zwiększeniu ilości bajtów przechowujących konkretną wartość bez zmiany tej wartości.

Czym jest promocja typu


```
8 bit                    => 16 bit
```

Czym jest promocja typu

```
8 bit                    => 16 bit

unsigned 00101001        => 00000000 00101001
unsigned 10101001        => 00000000 10101001
```

Czym jest promocja typu


```
8 bit                 => 16 bit

unsigned 00101001     => 00000000 00101001
unsigned 10101001     => 00000000 10101001


signed   00101001     => 00000000 00101001
signed   10101001     => 11111111 10101001
```

Czym jest konwersja typu?

Czym jest konwersja typu?

Konwersją typu nazywamy proces polegający na zmianie kodowania przechowywanej wartości, niekoniecznie zmieniając przy tym samą wartość.

Czym jest konwersja typu?


```
int32_t 42    => 00000000 00000000 00000000 00101010
```

Czym jest konwersja typu?


```
int32_t 42    => 00000000 00000000 00000000 00101010
float   42.0f => 01000010 00101000 00000000 00000000
```

Ukryty Integer Overflow

```
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << abs(INT_MIN) << std::endl;
    return 0;
}
```

?

Ukryty Integer Overflow

```
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << abs(INT_MIN) << std::endl;
    return 0;
}
```

```
Run:    program ×
/Users/senghe/Desktop/Prezka/cmake-build-debug/program
-2147483648

Process finished with exit code 0
```

Zwykła implementacja


```
unsigned abs(int value) {
    if (value < 0) {
        return -value;
    }
    return value;
}
```

Poprawna implementacja

```
unsigned safe_abs(int value) {
    if (value == INT_MIN) {
        throw E{};
    }
    return abs(value);
}
```

Big Integer

Big Integer

- Każda cyfra przechowywana jest jako kolejny element tablicy (niekoniecznie w formie dziesiętnej)

Big Integer

- Każda cyfra przechowywana jest jako kolejny element tablicy (niekoniecznie w formie dziesiętnej)
- Operacje na Big Integer-ach przypominają operacje pisemne stosowane w szkole podstawowej

Big Integer

- Każda cyfra przechowywana jest jako kolejny element tablicy (niekoniecznie w formie dziesiętnej)
- Operacje na Big Integer-ach przypominają operacje pisemne stosowane w szkole podstawowej
- Biblioteka standardowa C++ nie posiada implementacji dla Big Integer

Big Integer

- Każda cyfra przechowywana jest jako kolejny element tablicy (niekoniecznie w formie dziesiętnej)
- Operacje na Big Integer-ach przypominają operacje pisemne stosowane w szkole podstawowej
- Biblioteka standardowa C++ nie posiada implementacji dla Big Integer
- Biblioteka Boost posiada implementację Big Integer: boost/multiprecision/cpp_int

Big Integer

Zaleta: Ogarniamy duże liczby

Big Integer

Zaleta: Ogarniamy duże liczby

Wada: Procesory natywnie nie wspierają Big Integer-ów

Big Integer

Zaleta: Ogarniamy duże liczby

Wada: Procesory natywnie nie wspierają Big Integer-ów

Wada: Im większa liczba, tym wolniej się liczy

Ułamki bitowe

Ułamki bitowe

$14.0625_{dec} = ?_{bin}$

Ułamki bitowe

$14.0625_{dec} = ?_{bin}$

14 / 2 = 7	0
7 / 2 = 3	1
3 / 2 = 1	1
1 / 2 = 0	1

Ułamki bitowe

$14.0625_{dec} = 1110.?_{bin}$

Ułamki bitowe

$14.0625_{dec} = 1110.?_{bin}$

$0.0625 * 2 = 0.125$
$0.125 * 2 = 0.25$
$0.25 * 2 = 0.5$
$0.5 * 2 = 1.0$
$0 * 2 = 0$

Ułamki bitowe

$$14.0625_{dec} = 1110.00010_{bin}$$

Ułamki bitowe

$$14.0625_{dec} = 1110.00010000000_{bin}$$

Ułamki bitowe

- Stosując ułamki bitowe, to my decydujemy o ich dokładności

Ułamki bitowe

- Stosując ułamki bitowe, to my decydujemy o ich dokładności
- Na ułamkach bitowych można operować podobnie jak na liczbach całkowitych

Ułamki bitowe

- Stosując ułamki bitowe, to my decydujemy o ich dokładności
- Na ułamkach bitowych można operować podobnie jak na liczbach całkowitych
- Biblioteka standardowa C++ nie posiada implementacji dla liczb stałoprzecinkowych

Ułamki bitowe

- Stosując ułamki bitowe, to my decydujemy o ich dokładności
- Na ułamkach bitowych można operować podobnie jak na liczbach całkowitych
- Biblioteka standardowa C++ nie posiada implementacji dla liczb stałoprzecinkowych
- Przykładowa implementacja: http://johnmcfarlane.github.io/fixed_point

Liczby zmiennoprzecinkowe

Ile razy zostanie wykonana pętla?

```
#include <iostream>

int main() {
    float i=0.0f;
    do {
        std::cout << "Loop " << std::endl;
        i+=0.1f;
    } while (i != 1.0f);
}
```

Ile razy zostanie wykonana pętla?


```
#include <iostream>

int main() {
    float i=0.0f;
    do {
        std::cout << "Loop " << std::endl;
        i+=0.1f;
    } while (i != 1.0f);
}
```

Odpowiedź: to jest pętla nieskończona

Ale… dlaczego?

Niedokładne liczby zmiennoprzecinkowe

```
#include <iostream>
#include <iomanip>

int main() {
    std::cout << std::setprecision(10) << 0.1f << std::endl;
}
```

Na wyjściu: 0.1000000015

Co pojawi się na wyjściu?


```
#include <iostream>
#include <iomanip>

int main() {
    float amountOfProduct = 254.99f;
    unsigned long int quantity = 100000000;
    float amountOfCheckout = amountOfProduct*quantity;

    std::cout << std::setprecision(15) << amountOfCheckout << std::endl;
    return 0;
}
```

Co pojawi się na wyjściu?


```
#include <iostream>
#include <iomanip>

int main() {
    float amountOfProduct = 254.99f;
    unsigned long int quantity = 100000000;
    float amountOfCheckout = amountOfProduct*quantity;

    std::cout << std::setprecision(15) << amountOfCheckout << std::endl;
    return 0;
}
```

Odpowiedź: 25499000832

Co mówi standard…

“There are three floating-point types: float, double, and long double. The type double provides at least as much precision as float, and the type long double provides at least as much precision as double. The set of values of the type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set of values of the type long double. The value representation of floating-point types is implementation-defined. [Note: This document imposes no requirements on the accuracy of floating-point operations; see also [support.limits]. — end note] Integral and floating-point types are collectively called arithmetic types. Specializations of the standard library template **std::numeric_limits** shall specify the maximum and minimum values of each arithmetic type for an implementation.”

Źródło: http://eel.is/c++draft/basic.fundamental

Standard IEEE 754

Standard IEEE 754

$$0.1_{dec} = 00111101110011001100110011001101_{bin}$$

Standard IEEE 754

Znak Wykładnik Mantysa

$$0.1_{dec} = 0\,01111011\,10011001100110011001101_{bin}$$

Standard IEEE 754

	Znak	Wykładnik	Mantysa	
$0.1_{dec} =$	0	01111011	10011001100110011001101	$_{bin}$
float	1 bit	8 bitów	23 bity	

Standard IEEE 754

	Znak	Wykładnik	Mantysa

$0.1_{dec} = 00111101110011001100110011001101_{bin}$

	Znak	Wykładnik	Mantysa
float	1 bit	8 bitów	23 bity
double	1 bit	11 bitów	52 bity

Standard IEEE 754

	Znak	Wykładnik	Mantysa	

$$0.1_{dec} = 0\,01111011\,10011001100110011001101_{bin}$$

	Znak	Wykładnik	Mantysa	
float	1 bit	8 bitów	23 bity	bias = 127
double	1 bit	11 bitów	52 bity	bias = 1023

Float to bin

Standard IEEE 754

$-41.27_{dec} =$

Standard IEEE 754

Znak

$$-41.27_{dec} = 1$$

Standard IEEE 754

Znak

$-41.27_{dec} = 1$

41 / 2 = 20	1
20 / 2 = 10	0
10 / 2 = 10	0
5 / 2 = 2	1
2 / 2 = 1	0
1 / 2 = 0	1

Standard IEEE 754

Znak Mantysa

$$-41.27_{dec} = 1\ldots\ldots\ldots101001.$$

Standard IEEE 754

Znak Mantysa

$-41.27_{dec} = 1........101001.$

0.27 * 2 = 0.54
0.54 * 2 = 1.08
0.08 * 2 = 0.16
0.16 * 2 = 0.32
0.32 * 2 = 0.64
0.64 * 2 = 1.28
0.28 * 2 = 0.56
0.56 * 2 = 1.12
0.12 * 2 = 0.24
0.24 * 2 = 0.48
0.48 * 2 = 0.96
0.96 * 2 = 1.92
0.92 * 2 = 1.84
0.84 * 2 = 1.68
0.68 * 2 = 1.36
0.36 * 2 = 0.72
0.72 * 2 = 1.44
0.44 * 2 = 0.88
0.88 * 2 = 1.76
0.76 * 2 = 1.52
0.52 * 2 = 1.04
0.04 * 2 = 0.08

Standard IEEE 754

Znak Mantysa

$$-41.27_{dec} = 1\text{.........}101001.01000101000111101\textbf{0}111$$

Standard IEEE 754

Znak Mantysa

-41.27_{dec} = 1........101001.01000101000111011

2^5

Standard IEEE 754

Znak Mantysa

$-41.27_{dec} = 1\ldots\ldots\ldots 1.01001010001010001111011$

2^5

Standard IEEE 754

Znak Mantysa

$$-41.27_{dec} = 1\ldots\ldots\ldots01001010001010001111011$$

$$127+5 = 132_{dec} =$$

Standard IEEE 754

Znak Mantysa

$$-41.27_{dec} = 1\ldots\ldots\ldots01001010001010001111011$$

$$127+5 = 132_{dec} =$$

132	/ 2 = 66	0
66	/ 2 = 33	0
33	/ 2 = 16	1
16	/ 2 = 8	0
8	/ 2 = 4	0
4	/ 2 = 2	0
2	/ 2 = 1	0
1	/ 2 = 0	1

$$= 10000100_{bin}$$

Standard IEEE 754

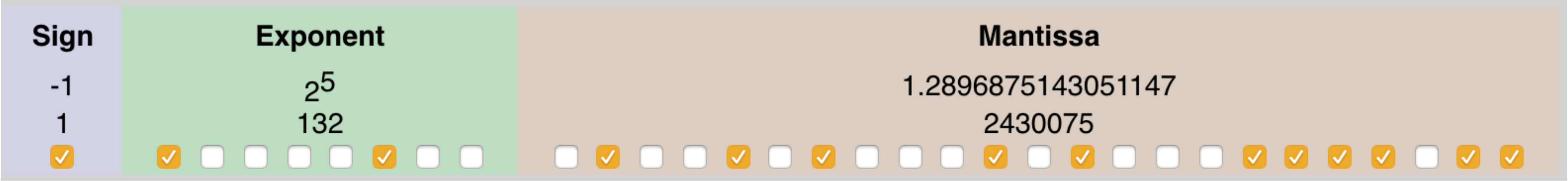
Znak Wykładnik Mantysa

$$-41.27_{dec} = 1\,10000100\,01001010001010001111011$$

Standard IEEE 754

Znak Wykładnik Mantysa

$$-41.27_{dec} = 11000010001001010001010001111011$$



Standard IEEE 754

Znak Wykładnik Mantysa

-41.27_{dec} = 11000010001001010001010001111011

Bin to float

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec} =$ 11000010001001010001010001111011

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec}$ = 11000010001001010001010001111011

$$L = (-1)^{Z} * M * 2^{K\text{-}bias}$$

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec} =$ 11000010001001010001010001111011

$$L = (-1)^{Z} * M * 2^{K\text{-bias}}$$

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec} =$ 1 10000100 01001010001010001111011

$$L = (-1)^{1} * M * 2^{K\text{-bias}}$$

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec} =$ 1 10000100 01001010001010001111011

$$L = (-1)^{1} * M * 2^{K-127}$$

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec}$ = 1 10000100 01001010001010001111011

$$L = (-1)^{1} * M * 2^{132-127}$$

Standard IEEE 754

Znak	Wykładnik	Mantysa

$?_{dec}$ = 11000010001001010001010001111011

$$L = (-1)^{1} * 1.2896875143051147 * 2^{132-127}$$

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec}$ = 1 10000100 01001010001010001111011

$L = -1 * 1.2896875143051147 * 2^5$

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec}$ = 11000010001001010001010001111011

L = -1.28968751430511474 * 32

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec} =$ 11000010001001010001010001111011

L = -41.2700004578

Standard IEEE 754

Znak Wykładnik Mantysa

$?_{dec}$ = 11000010001001010001010001111011

L = -41.2700004578

Standard IEEE 754

Znak	Wykładnik	Mantysa

$?_{dec}$ = 1 10000100 01001010001010001111011

L = -41.27000046

Standard IEEE 754

Wartości specjalne

```
NaN     =>    01111111111111111111111111111111
```

Standard IEEE 754

Wartości specjalne

```
NaN     =>    01111111111111111111111111111111

-INF    =>    11111111100000000000000000000000
```

Standard IEEE 754

Wartości specjalne

NaN => 01111111111111111111111111111111

-INF => 11111111100000000000000000000000

+INF => 01111111100000000000000000000000

Boost multiprecision

Boost multiprecision


```
// Sprawdzamy, czy możemy osiągnąć wartość 0.1
long double a = 0.1;
std::cout << std::setprecision(100) << a << std::endl;

cpp_dec_float_100 b("0.1");
std::cout << std::setprecision(100) << b << std::endl;
```

Boost multiprecision


```
long double a = 0.1;
std::cout << std::setprecision(100) << a << std::endl;

cpp_dec_float_100 b("0.1");
std::cout << std::setprecision(100) << b << std::endl;
```

Na wyjściu: 0.1000000000000000055511151231257827021181583404541015625
0.1

Boost multiprecision


```
cpp_dec_float_100 counter; // Domyślna wartość jest ustawiana na 0

do {
    std::cout << "Loop " << std::endl;
    counter += cpp_dec_float_100("0.1");
} while (counter != 1.0);
```

Boost multiprecision


```
cpp_dec_float_100 counter; // Domyślna wartość jest ustawiana na 0

do {
    std::cout << "Loop " << std::endl;
    counter += cpp_dec_float_100("0.1");
} while (counter != 1.0);
```

Pętla wykona się dokładnie 10 razy

Boost multiprecision


```
float amountOfProduct = 254.99f;
unsigned long int quantity = 100000000;
float amountOfCheckout = amountOfProduct*quantity;

std::cout << std::setprecision(30) << amountOfCheckout << std::endl;

cpp_dec_float_50 amountOfProduct2 = cpp_dec_float_50("254.99");
cpp_dec_float_50 quantity2 = cpp_dec_float_50("100000000");
cpp_dec_float_50 amountOfCheckout2 = cpp_dec_float_50(amountOfProduct2*quantity2);

std::cout << std::setprecision(30) << amountOfCheckout2 << std::endl;
```

Boost multiprecision


```
float amountOfProduct = 254.99f;
unsigned long int quantity = 100000000;
float amountOfCheckout = amountOfProduct*quantity;

std::cout << std::setprecision(30) << amountOfCheckout << std::endl;

cpp_dec_float_50 amountOfProduct2 = cpp_dec_float_50("254.99");
cpp_dec_float_50 quantity2 = cpp_dec_float_50("100000000");
cpp_dec_float_50 amountOfCheckout2 = cpp_dec_float_50(amountOfProduct2*quantity2);

std::cout << std::setprecision(30) << amountOfCheckout2 << std::endl;
```

Na wyjściu: 25499000832
25499000000

Decimal float a standard C++

Są propozycje dodania typów decimal floating-point do standardu:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3871.html

KONIEC