Poprawny, czytelny i wydajny kod w C - fakty i mity

Maciej Gajdzica

BITCONF

UCGOSU

O mnie

- Blog - ucgosu.pl
- Kanał YT - MaciekGajdzica
- Konferencje
- Gdańsk Embedded Meetup
- Kursy online - "C dla Zaawansowanych" i "TDD Embedded"

C - Trap Adventure

Wskaźniki

```c
#include "stdint.h"

uint8_t *u8_ptr;
uint32_t *u32_ptr;

_Static_assert(sizeof(u8_ptr) == sizeof(u32_ptr));

int main(void)
{
    return 0;
}
```

Czy ten program się skompiluje?

A) Nie

B) Tak

C) To zależy od platformy

https://godbolt.org/z/PnPaPxrhb

```c
#include <stdio.h>
#include <stdint.h>

int main(void)
{
    uint32_t *ptr = (uint32_t *)1000;
    ptr++;
    printf("%d\n", ptr);

    return 0;
}
```

Co ten program wypisze na konsolę?

A) 1000

B) 1001

C) 1004

D) To zależy od platformy

https://godbolt.org/z/9j8borvdE

Arytmetyka wskaźników

Dozwolone operacje:

- dodawanie stałej
- odejmowanie stałej
- odejmowanie wskaźnika tego samego typu

Arytmetyka wskaźników

Niedozwolone operacje:

- dodawanie wskaźnika
- mnożenie, dzielenie itd.

Arytmetyka wskaźników

- stałe są mnożone przez rozmiar typu
- zachowanie równoważne z indeksami tablicy

Obliczanie rozmiaru regionów pamięci

```c
#include <stdio.h>
#include <stdint.h>

#define MEMORY_START    0x20000000
#define MEMORY_END      0x20002000

#define MEMORY_SIZE1    ((MEMORY_END)-(MEMORY_START))

typedef uint16_t cellsize_t;

int main(void)
{
    int memory_size2 = (cellsize_t *)MEMORY_END - (cellsize_t *)MEMORY_START;
    int memory_size3 = (void *)MEMORY_END - (void *)MEMORY_START;

    printf("%d\n", MEMORY_SIZE1);
    printf("%d\n", memory_size2);
    printf("%d\n", memory_size3);

    return 0;
}
```

https://godbolt.org/z/ecWohz7aP

Typy `uintptr_t` i `ptrdiff_t`

```c
#include <stdio.h>
#include <stdint.h>
#include <stddef.h>

#define MEMORY_START    0x20000000
#define MEMORY_END      0x20002000

int main(void)
{
    ptrdiff_t memory_size = (uintptr_t)MEMORY_END - (uintptr_t)MEMORY_START;

    printf("%d\n", memory_size);

    return 0;
}
```

https://godbolt.org/z/TT5sojTzc

Typy `uintptr_t` i `ptrdiff_t`

- Znajdziemy je w `stddef.h`
- Rozmiar zależy od architektury
- Mamy gwarancję, że pomieszczą adresy
- Są portowalne

Przepełnienie liczb całkowitych

```c
#include <stdio.h>
#include <stdint.h>

void is_overflow(int32_t x)
{
    if (x + 1 < x)
    {
        printf("Overflow");
    }
    else
    {
        printf("No overflow");
    }
}

int main(void)
{
    is_overflow(INT32_MAX);
    return 0;
}
```

https://godbolt.org/z/37WdYW

Przepełnienie typów całkowitych

- Dla liczb bez znaku jest jasno zdefiniowane
- Dla liczb ze znakiem już nie
- Są możliwe różne realizacje liczb ujemnych

Kiedy używamy `signed int`?

- operacje arytmetyczne
- przydają nam się liczby ujemne
- overflow jest niepożądany
- kody błędów - potrzebujemy liczby ujemne

Kiedy używamy `unsigned int`?

- iteratory, liczniki, timestampy - liczymy od zera i mogą się przepełnić
- operacje bitowe - bit znaku nie powoduje dziwnego zachowania

Operacje bitowe

```c
#include <stdint.h>
#include <stdio.h>

int main(void)
{
    int32_t mask1 = 0xFFFFFFFF;
    uint32_t mask2 = 0xFFFFFFFF;

    printf("0x%08X\n", mask1 >> 5);
    printf("0x%08X\n", mask2 >> 5);

    return 0;
}
```

https://godbolt.org/z/GExP59

```c
#include <stdint.h>
#include <stdio.h>

int main(void)
{
    int16_t mask1 = 0x8000;
    int32_t mask2 = 0x8000;

    printf("0x%08X\n", mask1);
    printf("0x%08X\n", mask2);

    mask2 = mask1;

    printf("0x%08X\n", mask2);

    return 0;
}
```

https://godbolt.org/z/MrE4GM

```c
#include <stdio.h>
#include <stdint.h>

void func(uint16_t a, uint16_t b)
{
    uint32_t x;
    x = a * b;

    if(x > 0x80000000)
        printf("%u is more then %u\n", x, 0x80000000);
    else
        printf("%u is less or equal then %u\n", x, 0x80000000);
}

int main(void)
{
    func(0xFFFF, 0xFFFF);

   return 0;
}
```

https://godbolt.org/z/zaWbGaMT4

Co się wydarzyło?

- `uint16_t` został niejawnie skonwertowany na `int32_t`
- Kompilator zoptymalizował ifa
- Rzutowanie `int32_t` na `uint32_t` inaczej interpretuje bit znaku
- I przypadkiem daje dobry wynik
- A przy okazji overflow przy `signed int` to UB


8DIGIT CMB801-BK
2POWER
MC
OFF
MR
√
CE
M−
%
C
7
8
9
M+
+/−
ON
4
5
6
÷
1
2
3
×
0
•
=
−
+

Dlaczego to tak działa?

- Jeśli premiujemy znak spowodujemy inne problemy
- `uint16_t` rzutuje się do `uint32_t`
- to `uint16_t` * `uint16_t` * `int32_t` rzutuje się do `uint32_t`

Dobre praktyki

- ustalmy dla projektu domyślny typ operacji arytmetycznych
- najlepiej natywny dla architektury CPU int ze znakiem
- minimalizujemy w ten sposób dziwne zachowania
- a przy okazji obliczenia są wtedy wydajne

Stringi w C

Czy te dwa stringi się czymś różnią?

```
char *str1 = "Hello world!";
char str2[] = "Hello world!";
```

```c
#include <stdio.h>
#include <stdint.h>

char *str1 = "Hello world!";
char str2[] = "Hello world!";

int main(void)
{
    str1[3] = '$';

    printf("%s\n", str1);

    return 0;
}
```

https://godbolt.org/z/xxzbW3YT6

`char *` - wskaźnik na łańcuch znaków

- może być zapisany w pamięci tylko do odczytu
- ale nie musi
- w każdym razie nie możemy go modyfikować
- dlatego lepiej to jasno napisać - `const char *`

`char []` - tablica znaków

- Jest zapisywana do tablicy alokowanej w RAMie
- Można ją modyfikować
- Chyba, że dodamy w deklaracji `const`

Stringi jako argumenty funkcji

```
char * strcpy (char * destination, const char * source);
```

https://godbolt.org/z/nGTreKz1h

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

char *str1 = "Hello world!";
char str2[] = "Hello world!";

int main(void)
{
    strcpy(str2, str1);

    return 0;
}
```

Substringi

```c
#include <stdio.h>
#include <stdint.h>

char *str1 = "Hello world!";
char *str2 = "world!";

int main(void)
{
    printf(str1);
    printf(str2);

    return 0;
}
```

https://godbolt.org/z/ca6szj4ch

Tego się nie spodziewacie

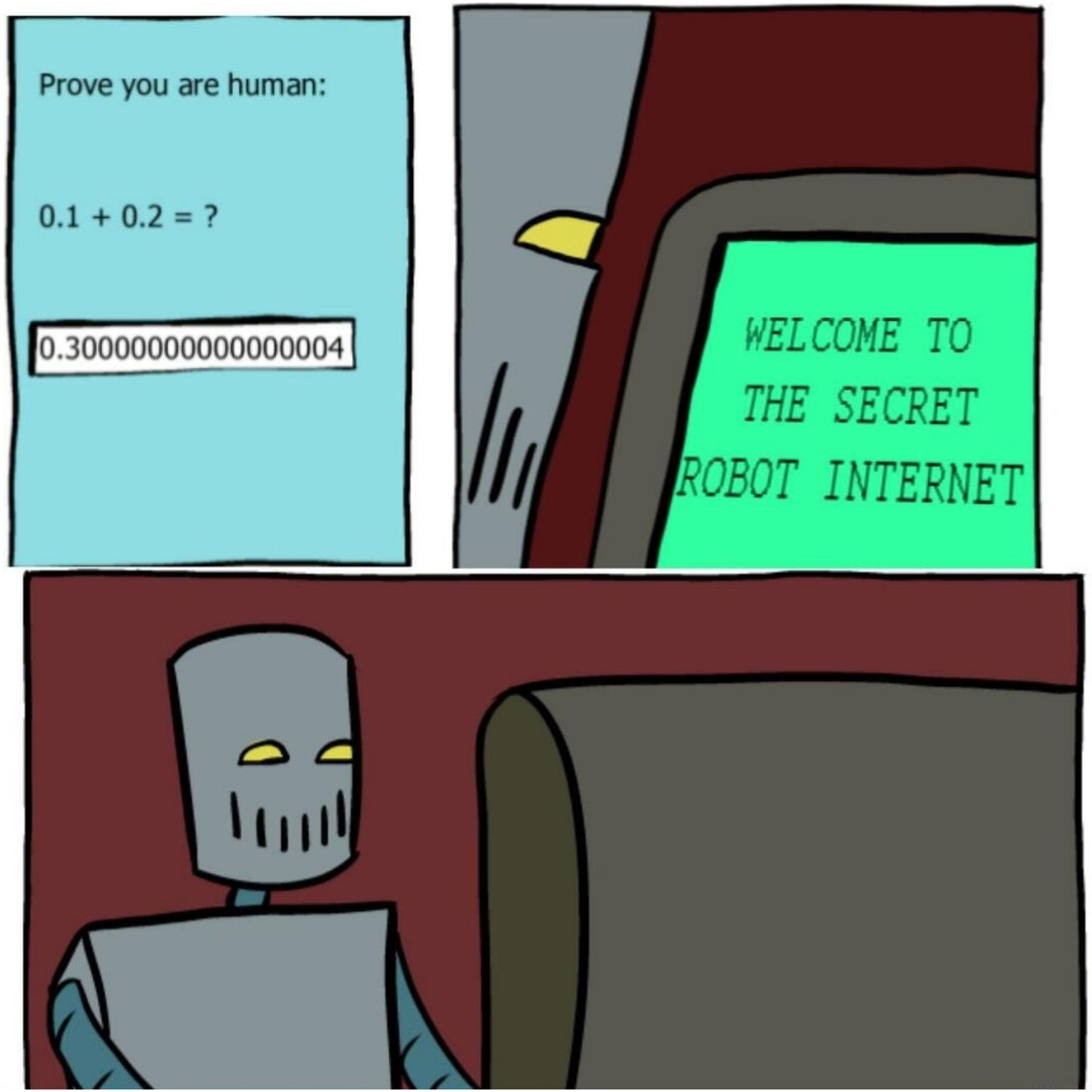
```c
#include <stdio.h>
#include <stdint.h>

const char * get_day(int day_id)
{
return "Mon\0Tue\0Wed\0Thu\0Fri\0Sat\0Sun" + day_id*4;
}

int main()
{
    for (int i = 0; i < 7; i++)
        printf("%s\n", get_day(i));
}
```

https://godbolt.org/z/xjd7Ydf9a

Floaty


Prove you are human:
0.1 + 0.2 = ?
0.30000000000000004
WELCOME TO
THE SECRET
ROBOT INTERNET

```c
#include <stdio.h>
#include <stdint.h>

int main(void)
{
    float a = 0.5;
    float b = a + 0.3;
}
```

https://godbolt.org/z/46jYjKvqa

Domyślny typ zmiennoprzecinkowy to `double`!

Co robić, jak żyć?

- Suffix `f` wymusza typ float
- `-Wdouble-promotion` - warning gcc kiedy float rzutujemy na double
- `-fsingle-precision-constant` - stałe `float` bez suffixu `f`

https://godbolt.org/z/q6rjsP

Funkcje z `math.h`

```c
#include <stdio.h>
#include <stdint.h>
#include <math.h>

int main(void)
{
    float a = 0.5;
    float b = a + 0.3;

    sqrt(a);
}
```

Funkcje z `math.h` mają trzy wersje (od `C99`)

- `sqrt` - pierwiastek na double
- `sqrtf` - pierwiastek na float
- `sqrtl` - pierwiastek na long double

Tak samo wszystkie inne funkcje:
https://en.cppreference.com/w/c/numeric/math

`tgmath.h` - Type Generic math

- `c99` - kompilator implementuje po swojemu
- `c11` - macro `_Generic`

Inline

Jaki był zamysł?

- Zamiast wołania funkcji - wklejamy jej zawartość
- Kompilator nie robi operacji skoku - zysk wydajności
- Kompilator dalej sprawdza błędy i warningi
- Łatwiejszy debug

Jak to wygląda w praktyce

- `inline` to dla kompilatora tylko sugestia
- nie gwarantuje usunięcia skoku
- kompilator może wklejać funkcje bez modyfikatora `inline`

Ale tak naprawdę jest jeszcze lepiej!

`inline` to optymalizacja

- Nie działa na `O0`
- Atrybut `__attribute__((always_inline))`

Ciekawe przypadki

- wołanie funkcji inline z innych plików
- używanie zmiennych statycznych
- wskaźniki na funkcje

Nic nie zyskujemy a możemy nieźle namieszać

Wnioski

- Warto wnikliwie poznać język
- Dobre praktyki pomagają unikać wad języka
- Optymalizacja nie psuje kodu, po prostu C działa nieintuicyjnie
- Warto czasem popatrzeć w asembler
- Nie warto optymalizować ręcznie na siłę

WIĘCEJ NA:

cdlazaawansowanych.pl/bitconf-2022

UCGOSU