

# Jak pisać w C i nie zwariować

Krzysztof Kuźnik

Mateusz Kwiatkowski



# Dlaczego pisać w C, skoro jest...

- ▶ Python, Ruby, JavaScript, ...
  - ▶ performance
  - ▶ bolączki dynamicznego typowania, zwłaszcza w większych projektach
- ▶ Java, Scala, C#, ... (JVM/.NET)
  - ▶ performance?
    - ▶ mit (zwykle)
  - ▶ wsparcie dla akceleracji sprzętowej
  - ▶ niedeterministyczne zarządzanie pamięcią
    - ▶ systemy czasu rzeczywistego
  - ▶ platforma jest DUŻA
    - ▶ JRE - ~150 MB

# Dlaczego pisać w C, skoro jest...

- ▶ C++, Rust, ...
  - ▶ Jak skompilować kompilator Rusta na egzotyczną architekturę?
  - ▶ ROZMIAR
    - ▶ `libstdc++` - 1,5 MB
      - ▶ Rozwinięcia szablonów...?
    - ▶ Rust `libstd` - 4 MB
    - ▶ `eglibc` - 1,8 MB - a powyższe i tak jej wymagają!
      - ▶ `uClibc` - 560 KB
      - ▶ `dietlibc` - 185 KB

# C "studenckie"

- ▶ Programy w jednym pliku
- ▶ BFG (big fucking globals)
- ▶ C++ (C with classes)
- ▶ Zalegizować STL? A komu to potrzebne?
- ▶ Printf debugging
- ▶ A może jednak Java?
- ▶ Przecież C i C++ to jest to samo
- ▶ C jest dla linuxowych freaków bo już nawet naukowcy używają Pythona

# C "profesjonalne"

- ▶ Narzędzia
- ▶ Budowanie projektu
- ▶ Zarządzanie pamięcią
- ▶ Unikanie pułapek
- ▶ Wzorce projektowe

# Narzędzia prawdziwego programisty

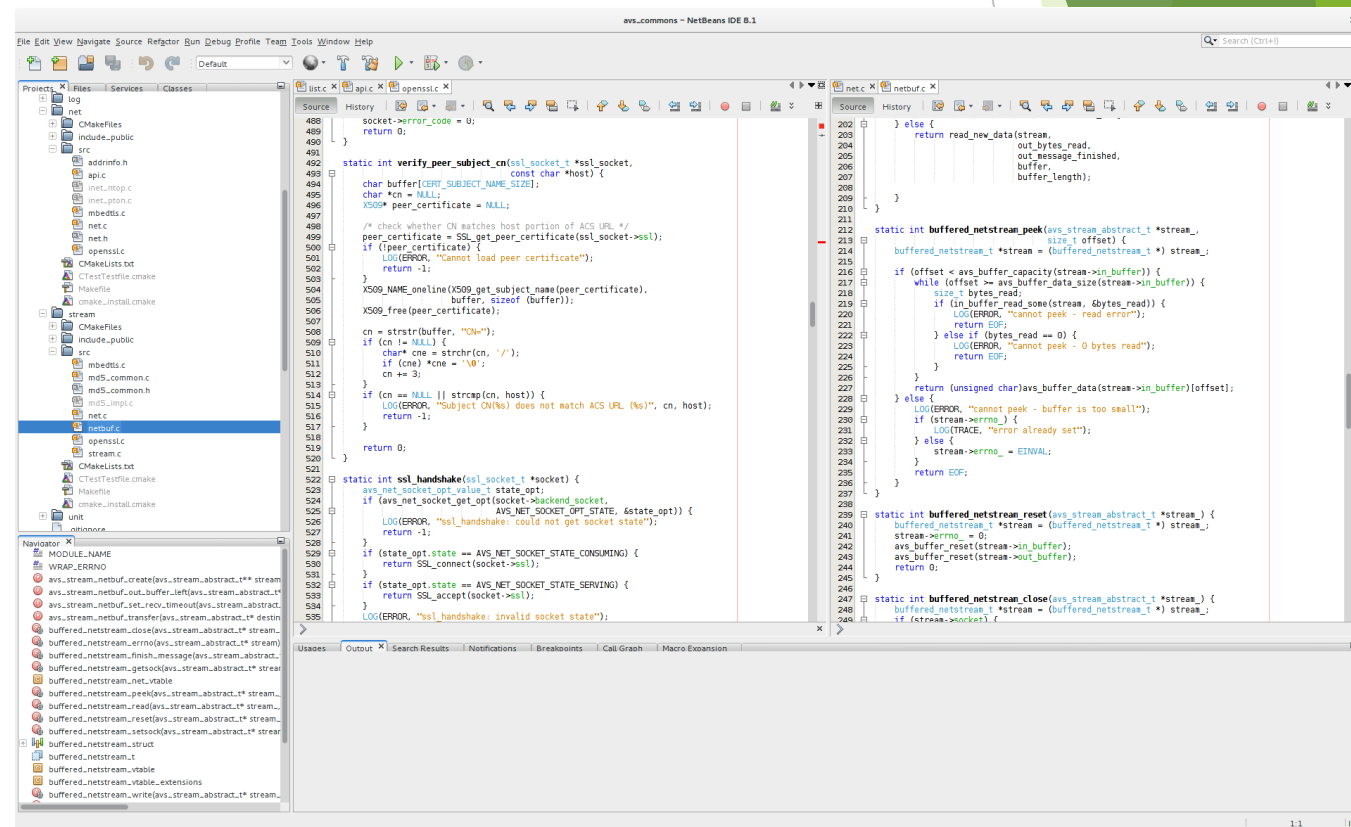
# Narzędzia prawdziwego programisty: Miejsce tworzenia

- ▶ W projektach studenckich (2-3 jednostki translacji) można pisać w gedit/Notepad++ i kompilować z konsoli...
- ▶ W prawdziwym życiu prawdziwych ludzi przyda się IDE
  - ▶ vim / Emacs / ... + masa pluginów
  - ▶ Visual Studio - skomplikowana licencja, tylko Windows
  - ▶ Eclipse + CDT
  - ▶ NetBeans



# NetBeans jako IDE do C/C++

- ▶ Obsługuje standardowe formaty projektów (Make, CMake, ...)
- ▶ Nawigacja po kodzie
  - ▶ Ctrl+klik, Alt+F7
- ▶ GUI do debuggera
- ▶ Pro tipy:
  - ▶ Przebuduj kod z poziomu IDE, żeby odświeżyć strukturę projektu
  - ▶ Łatwy podgląd nagłówków bibliotecznych, rozwijanie makr
- ▶ Jakież wady?
  - ▶ Żabkuje :(





# Narzędzia prawdziwego programisty:

## Plac budowy

- ▶ Wywoływanie kompilatora z terminala szybko staje się problematyczne...
- ▶ Zarządzanie projektem przez IDE?
  - ▶ Utrudniamy życie ludziom, którzy będą chcieli po prostu ściągnąć i zbudować projekt
  - ▶ Np. wykładowcom (ಠ\_ಠ)
- ▶ Surowe Makefile
  - ▶ Ograniczone możliwości
  - ▶ Nie wykrywa zmian w nagłówkach
  - ▶ Problem z wyszukiwaniem bibliotek
- ▶ GNU autotools
- ▶ CMake

```
LIBS = -ldl -lm

%.o: %.c
        $(CC) -c $^ -o $@

my_app: main.o logic.o ui.o
        $(CC) $^ $(LIBS) -o $@
```

# CMake

- ▶ <https://cmake.org/>
- ▶ Tutorial: <https://cmake.org/cmake-tutorial/>
- ▶ Nakładka na make
  - ▶ `cmake .` - generuje Makefile (lub projekt VS, ...)
  - ▶ `make` - właściwe budowanie
- ▶ Opis projektu szybko się rozrasta
  - ▶ CMake jest w pełni funkcjonalnym językiem skryptowym... ๖\_๖
  - ▶ Mimo wszystko, stosunkowo dobrze się skaluje
- ▶ Natywnie wspierany przez CLiona
- ▶ Stosunkowo dobrze działa z Eclipse i NetBeans
  - ▶ Najlepiej najpierw stworzyć szkielet projektu CMake i później importować!

```
cmake_minimum_required(VERSION 2.6)
project(Tutorial)
add_executable(Tutorial
               main.c logic.c ui.c)
```

# Narzędzia prawdziwego programisty: Gabinet postępowania kryzysowego

- ▶ Analiza statyczna
- ▶ Analiza dynamiczna
- ▶ Debugger



# Analiza statyczna

- Clang Static Analyzer:  
**scan-build**
- cmake  
-DCMAKE\_C\_COMPILER=  
{...}/ccc-analyzer .  
  
scan-build make
- Wykrywa wiele typowych błędów logicznych przed uruchomieniem programu

```
1464 n2s(p, payload);
1465 pl = p;
1466
1467 if (s->msg_callback)
1468     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
1469                     &s->s3->rrec.data[0], s->s3->rrec.length,
1470                     s, s->msg_callback_arg);
1471
1472 if (hbtype == TLS1_HB_REQUEST)
1473 {
1474     unsigned char *buffer, *bp;
1475     int r;
1476
1477     /* Allocate memory for the response, size is 1 byte
1478      * message type, plus 2 bytes payload length, plus
1479      * payload, plus padding
1480      */
1481     buffer = OPENSSL_malloc(1 + 2 + payload + padding);
1482     bp = buffer;
1483
1484     /* Enter response type, length and copy payload */
1485     *bp++ = TLS1_HB_RESPONSE;
1486     s2n(payload, bp);
1487     memcpy(bp, pl, payload);

```

1 Taking false branch →

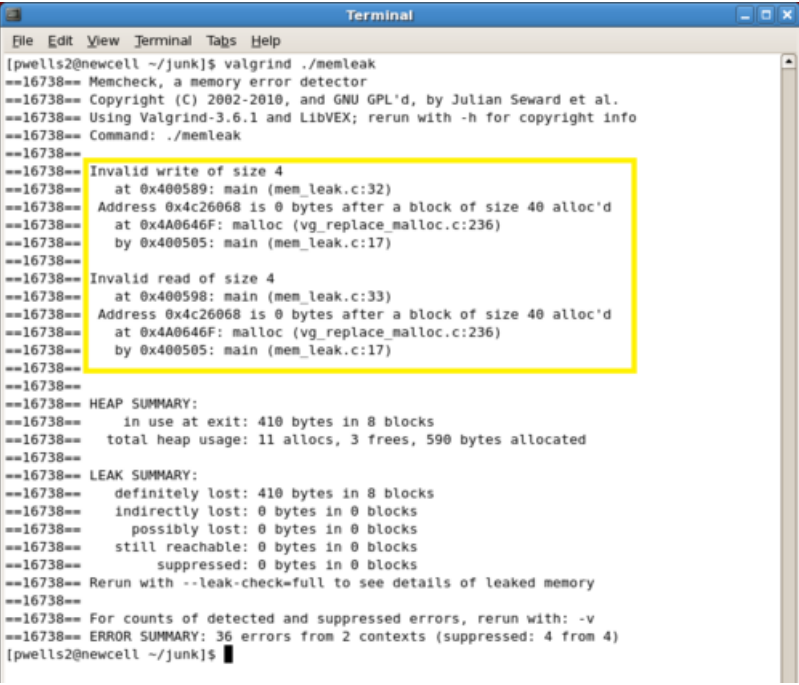
2 ← Assuming 'hbtype' is equal to 1 →

3 ← Taking true branch →

4 ← Tainted, unconstrained value used in memcpy size

# Analiza dynamiczna

- ▶ **Valgrind**
- ▶ Różne narzędzia, podstawowe: Memcheck
- ▶ `valgrind ./prog`
- ▶ `valgrind --leak-check=full ./prog`
- ▶ Inne narzędzia:
  - ▶ `valgrind --tool=...`
  - ▶ Helgrind - wykrywa błędy w programach wielowątkowych
  - ▶ Massif - profilowanie zużycia pamięci
  - ▶ Callgrind - profilowanie kodu
  - ▶ ...
- ▶ Alternatywy:
  - ▶ `clang -fsanitize=...`



```
Terminal
File Edit View Terminal Tabs Help
[pwells2@newcell ~/junk]$ valgrind ./memleak
==16738== Memcheck, a memory error detector
==16738== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==16738== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==16738== Command: ./memleak
==16738==
==16738== Invalid write of size 4
==16738== at 0x400589: main (mem_leak.c:32)
==16738== Address 0x4c26068 is 0 bytes after a block of size 40 alloc'd
==16738== at 0x4A0646F: malloc (vg_replace_malloc.c:236)
==16738== by 0x400505: main (mem_leak.c:17)
==16738==
==16738== Invalid read of size 4
==16738== at 0x400598: main (mem_leak.c:33)
==16738== Address 0x4c26068 is 0 bytes after a block of size 40 alloc'd
==16738== at 0x4A0646F: malloc (vg_replace_malloc.c:236)
==16738== by 0x400505: main (mem_leak.c:17)
==16738==
==16738== HEAP SUMMARY:
==16738== in use at exit: 410 bytes in 8 blocks
==16738== total heap usage: 11 allocs, 3 frees, 590 bytes allocated
==16738==
==16738== LEAK SUMMARY:
==16738== definitely lost: 410 bytes in 8 blocks
==16738== indirectly lost: 0 bytes in 0 blocks
==16738== possibly lost: 0 bytes in 0 blocks
==16738== still reachable: 0 bytes in 0 blocks
==16738== suppressed: 0 bytes in 0 blocks
==16738== Rerun with --leak-check=full to see details of leaked memory
==16738==
==16738== For counts of detected and suppressed errors, rerun with: -v
==16738== ERROR SUMMARY: 36 errors from 2 contexts (suppressed: 4 from 4)
[pwells2@newcell ~/junk]$
```

# Debugger

- ▶ Gdb
- ▶ "Printf debugging" nie zawsze jest złe, ale debugger daje więcej możliwości
- ▶ Surowy gdb jest trochę toporny
  - ▶ UI w IDE
  - ▶ gdb-dashboard
    - <https://github.com/cyrus-and/gdb-dashboard>
- ▶ Dodatkowe narzędzia:
  - ▶ gdbserver - zdalne debugowanie
  - ▶ valgrind --vgdb
  - ▶ rr

```
gdb
Output/messages
0x0000000100000f2a 7 for (i = 0; i < n; i++) {
Source
2
3 void fun(int n, char *data[])
4 {
5     int i;
6
7     for (i = 0; i < n; i++) {
8         printf("%d: %s\n", i, data[i]);
9     }
10 }
11
12 int main(int argc, char *argv[])
Assembly
0x0000000100000f18 b0 00 fun+56 mov $0x0,%al
0x0000000100000f1a e8 4b 00 00 00 fun+58 callq 0x100000f6a
0x0000000100000f1f 89 45 e8 fun+63 mov %eax,-0x18(%rbp)
0x0000000100000f22 8b 45 ec fun+66 mov -0x14(%rbp),%eax
0x0000000100000f25 05 01 00 00 00 fun+69 add $0x1,%eax
0x0000000100000f2a 89 45 ec fun+74 mov %eax,-0x14(%rbp)
0x0000000100000f2d e9 c4 ff ff ff fun+77 jmpq 0x100000ef6 <fun+22>
0x0000000100000f32 48 83 c4 20 fun+82 add $0x20,%rsp
0x0000000100000f36 5d fun+86 pop %rbp
0x0000000100000f37 c3 fun+87 retq
Threads
[1] id 4355 from 0x0000000100000f2a in fun+74 at scrot.c:7
Stack
[0] from 0x0000000100000f2a in fun+74 at scrot.c:7
arg n = 3
arg data = 0x7fff5fbffb60
loc i = 1
[1] from 0x0000000100000f62 in main+34 at scrot.c:14
arg argc = 3
arg argv = 0x7fff5fbffb60
(no locals)
Registers
rax 0x0000000000000002 rbx 0x0000000000000000 rcx 0x0000010000000203
rdx 0x0000020000000200 rsi 0x0000000000012068 rdi 0x00007fff79e86118
rbp 0x00007fff5fbffb20 rsp 0x00007fff5fbffb00 r8 0x0000000000000040
r9 0x00007fff79e86110 r10 0xffffffffffffffff r11 0x0000000000000246
r12 0x0000000000000000 r13 0x0000000000000000 r14 0x0000000000000000
r15 0x0000000000000000 rip 0x0000000100000f2a eflags [ TF IF ]
cs 0x0000002b ss <unavailable> ds <unavailable>
es <unavailable> fs 0x00000000 gs 0x00000000
Expressions
[1] data[i] = 0x7fff5fbffc0 "hello"
Memory
0x00007fff5fbffb0c 01 00 00 00 60 fb bf 5f ff 7f 00 00 00 00 00 00 ....._.b...
0x00007fff5fbffb1c 03 00 00 00 40 fb bf 5f ff 7f 00 00 62 0f 00 00 ...@.....b...
0x00007fff5fbffb2c 01 00 00 00 60 fb bf 5f ff 7f 00 00 03 00 00 00 .....P.....
0x00007fff5fbffb3c 00 00 00 50 fb bf 5f ff 7f 00 00 ad 15 f7 9c .....P.....
0x00007fff5fbffc0 68 65 6c 6c 6f 00 47 44 42 00 54 45 52 4d 5f 50 hello.GDB.TERM_P
History
$$1 = 63
$$0 = {[0] = 0x7fff5fbffc0 "hello", [1] = 0x7fff5fbffc6 "GDB"}
>>> |
```

# Inne narzędzia

- ▶ Profiler
- ▶ Unit testing library
- ▶ Fuzzer
- ▶ Doxygen

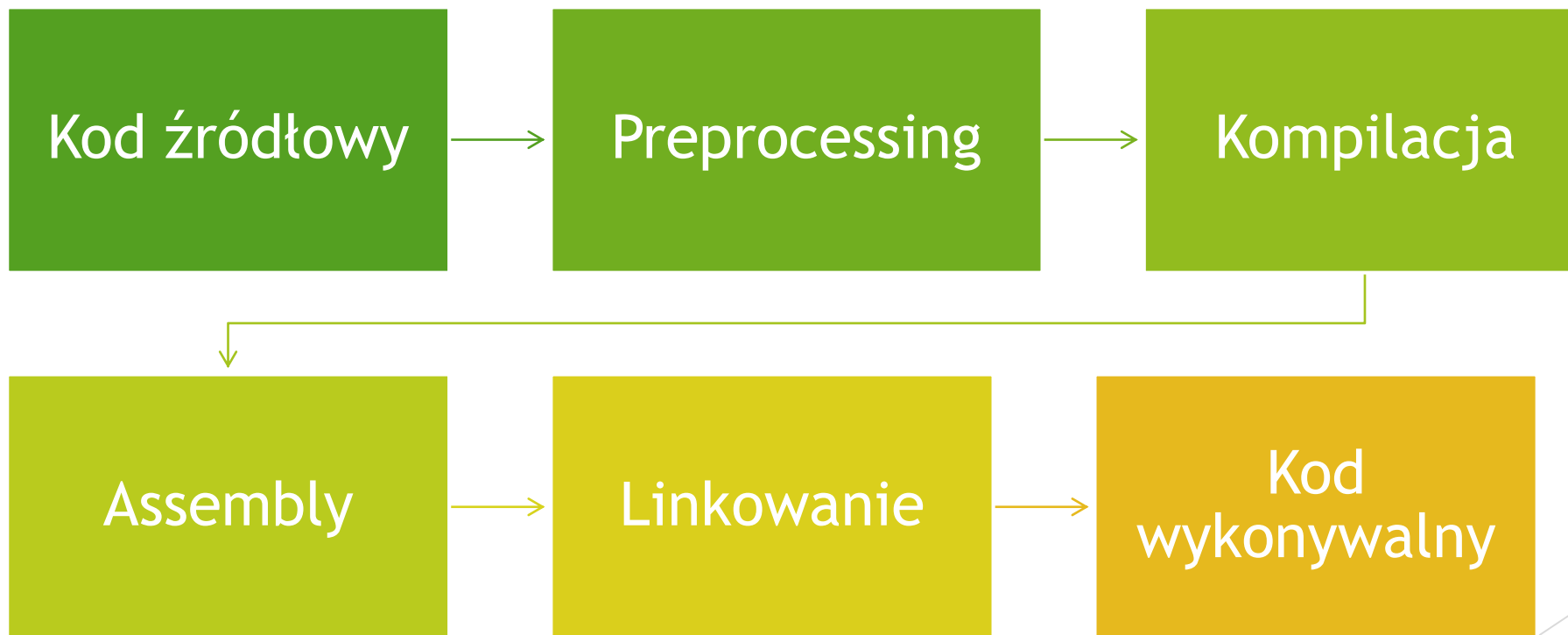
# Budowanie projektu

A może by tak podzielić program na wiele plików?

 O'RLY?



# Etapy "kompilacji"



# Jednostka translacji

- ▶ Plik źródłowy po preprocessingu
- ▶ Czasem zwana też jednostką kompilacji
- ▶ Podstawa dla działania kompilatora
- ▶ Co się dzieje w trakcie preprocessingu?

# Pliki nagłówkowe

```
// something is missing here  
  
int main(void) {  
    printf("Header files\n");  
    return 0;  
}
```

# Pliki nagłówkowe

```
#include <stdio.h>

int main(void) {
    printf("Header files\n");
    return 0;
}
```

# Pliki nagłówkowe

```
int printf(const char *, ...);

int main(void) {
    printf("Header files\n");
    return 0;
}
```

# Pliki nagłówkowe

- ▶ Deklaracja musi być
- ▶ Definicja to też deklaracja
- ▶ Definicja symbolu musi być zgodna z deklaracją
- ▶ Deklaracje w jednym miejscu
- ▶ gcc -E

# Kod obiektowy

- ▶ Kompilacja + assembly
- ▶ Cross-compiling
- ▶ Kontrola typów
- ▶ Optymalizacje
- ▶ `gcc -S -masm=intel`
- ▶ Kod maszynowy gotowy do linkowania
- ▶ `gcc -c`

# Linkowanie

- ▶ Łączenie plików obiektowych
- ▶ Łączenie symboli
- ▶ Dołączenie bibliotek
- ▶ Setup środowiska



# Najprostszy scenariusz

```
projekt  
├── main.c  
├── module.c  
└── module.h
```

# module.h

```
#ifndef _PROJEKT_MODULE_H_  
#define _PROJEKT_MODULE_H_  
  
void did_leo_get_his_oscar(void) ;  
  
#endif /* _PROJEKT_MODULE_H_ */
```

# module.c

```
#include <stdio.h>

#include <module.h>

void did_leo_get_his_oscar(void) {
    printf("Yes\n");
}
```

# main.c

```
#include <module.h>

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

# And the Oscar goes to...

```
$ gcc -Wall -pedantic -std=c99 -I. -o did_leo_get_his_oscar main.c module.c
```

```
$ ./did_leo_get_his_oscar  
Yes
```



# Enkapsulacja!

- ▶ C nie ma namespace'ów ani przeladowania nazw funkcji
- ▶ `static` FTW!!!
- ▶ "prywatne" struktury

## static (1/2)

```
int counter(void) {  
    static int x = 0;  
    return x++;  
}
```

## static (2/2)

```
static const char *IMPORTANT_TEXT = "HTTP";
static const uint32_t DEFAULT_VALUE = 8;

static uint32_t some_internal_function(void) {
    return DEFAULT_VALUE;
}

extern uint32_t module_public_function(void);

uint32_t module_public_function(void) {
    return some_internal_function();
}
```



# Przeźroczyste typy danych

Jak ograniczyć widoczność pól struktury?

# main.c

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

int main(void) {
    student_t *my_student = NULL;
    if (!create_student(&my_student)) {
        printf("success");
        free_student(my_student);
    } else {
        printf("failure");
    }
    return 0;
}
```

# module.h

```
#ifndef _PROJEKT_MODULE_H_  
#define _PROJEKT_MODULE_H_  
  
struct student;  
typedef struct student student_t;  
  
int create_student(student_t **);  
void free_student(student_t *);  
  
#endif /* _PROJEKT_MODULE_H_ */
```

# module.c

```
#include <module.h>

struct student {
    char name[257];
};

int create_student(student_t **student) {
    /* memory allocation & initialization */
}

void free_student(student_t *student) {
    free(student);
}
```

# Zarządzanie pamięcią

Czy to jest aż tak trudne?

# Ciekące zasoby

- ▶ Pamięć
- ▶ Deskryptory plików
  - ▶ Pliki
  - ▶ Sockety
  - ▶ Pipe'y
- ▶ Wątki

“

**ZAWSZE** sprzątaj swój bałagan

”

Złota zasada nr 1

```
int read_data(void) {
    FILE *file1 = fopen("plik1", "r");
    FILE *file2 = fopen("plik2", "r");
    int result = -1;

    if (!file1 || !file2) {
        goto read_data_cleanup;
    } else {
        /* inne operacje na file1 i file2 */
        result = 0;
    }

read_data_cleanup:
    if (file1) {
        fclose(file1);
    }
    if (file2) {
        fclose(file2);
    }
    return result;
}
```



“

Twoje funkcje powinny mieć  
tylko jedno słowo kluczowe  
**return**

”

Złota zasada nr 2

Nie jest to konieczne, ale jest bardzo pomocne w analizie kodu.

```
int read_data(void) {
    FILE *file1 = fopen("plik1", "r");
    FILE *file2 = fopen("plik2", "r");
    int result = -1;

    if (!file1 || !file2) {
        goto read_data_cleanup;
    } else {
        /* inne operacje na file1 i file2 */
        result = 0;
    }

read_data_cleanup:
    if (file1) {
        fclose(file1);
    }
    if (file2) {
        fclose(file2);
    }
    return result;
}
```

“

# Zawsze minimalizuj zasięg dostępu do swoich danych

”

Złota zasada nr 3

Bloki kodu mają sens nie tylko przy instrukcjach sterujących.  
Jeżeli alokujesz jakiś zasób to staraj się go zwolnić w tej samej funkcji.  
(nie dotyczy to funkcji, które są nastawione na tworzenie lub usuwanie "instancji")

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern, layered effect on the right side of the slide.

# True bugs wait...

<http://natashenka.ca/posters/>

I have a lot to look forward to in life. That's why using  
`strcat` isn't an option. Only abstaining from `strcat`  
can 100% protect me from overflows and memory  
disclosures

`strcat` just isn't  
worth it



[natashenka.ca/strcat](https://natashenka.ca/strcat)



Canadian Joke  
Council

*True Bugs Wait ♡*

@natashenka

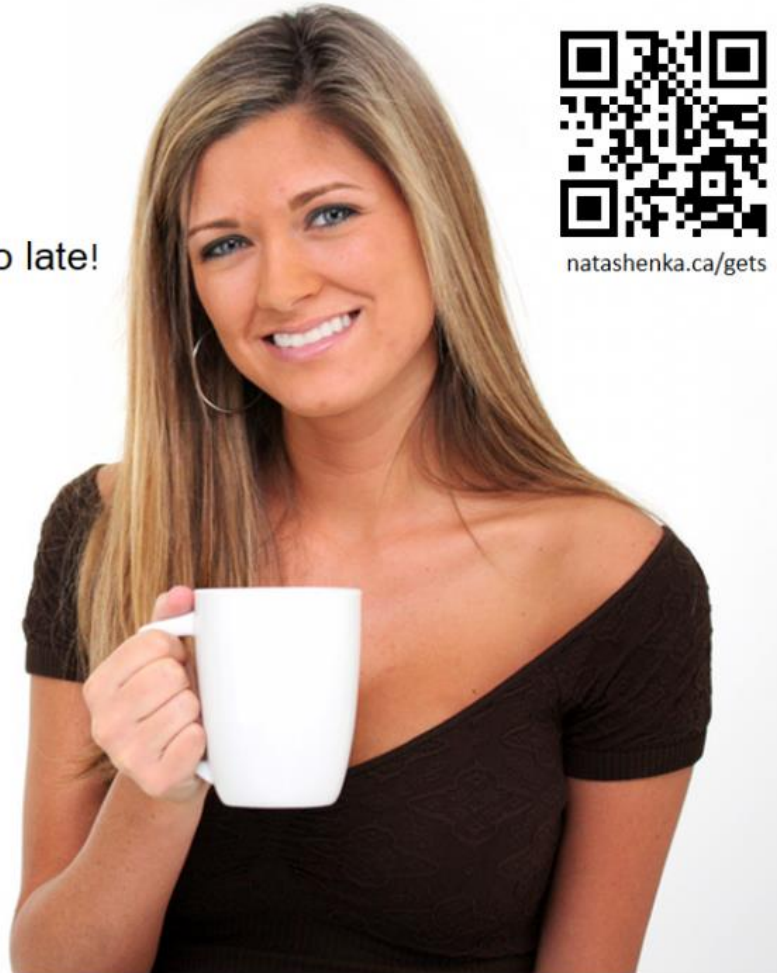
#truebugswait

Just because I used `gets` before doesn't mean I have to use `gets` again. It's my choice! Now that I know more about `gets`, I'm waiting for a standard input function I can trust for life

It's never too late!



[natashenka.ca/gets](http://natashenka.ca/gets)



# Bezpieczeństwo

- ▶ `strncat`
- ▶ `strncpy`
- ▶ `snprintf`
- ▶ `strncmp`
- ▶ `NULL` terminator
- ▶ Zawsze inicjalizuj zmienne (tablice, struktury, wskaźniki)

# Przybywacie z krainy Pascala?

```
int main() {  
    int i;  
    int tab[10];  
    for (i = 1; i <= 10; ++i) {  
        tab[i] = 0;  
    }  
    return 0;  
}
```



# Dawno, dawno temu, na serwerach KI...

- Co jest nie tak z tym kodem...?

```
void put_u32(char *buf,  
             size_t offset,  
             uint32_t val) {  
    *(uint32_t *) &buf[offset] = val;  
}
```

# Dawno, dawno temu, na serwerach KI...

► Rozwiązanie!

```
void put_u32(char *buf,  
            size_t offset,  
            uint32_t val) {  
    memcpy(&buf[offset],  
          &val, sizeof(val));  
}
```

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic look.

# C99 jest fajne!

...i już prawie pełnoletnie ;)

# Nie używaj języka starszego od siebie!

- ▶ Wiele kompilatorów wciąż domyślnie używa standardu C89
  - ▶ On ma 26 lat!
- ▶ C99 jest już dostępne prawie wszędzie!
  - ▶ GCC od... bardzo dawna
  - ▶ Clang od początku istnienia
  - ▶ Visual C++... hmm... cóż... :(
- ▶ C11 coraz szerzej dostępne
- ▶ Dużo wygodnych usprawnień
  - ▶ Deklaracje zmiennych w dowolnym miejscu
  - ▶ Lokalne tablice o "zmiennym" rozmiarze
  - ▶ Wygodniejsze definicje stałych tablicowych i strukturalnych
  - ▶ ...

# C99 jest fajne!

## Lokalne tablice o "zmiennym" rozmiarze

- ▶ NIE chodzi o tablice dynamiczne

- ▶ Przykład:

```
▶ void read_command(FILE *file) {  
    size_t command_size;  
    fread(&command_size, sizeof(command_size), 1, file);  
    char command[command_size];  
    // ...  
}
```

- ▶ Dane są alokowane na stosie

- ▶ wydajniejsze niż malloc()/free()

- ▶ Nie przesadzajmy z rozmiarem danych!

- ▶ Jeśli wyjdziemy poza rozmiar stosu (kilka MB), błąd będzie bardzo trudny do wykrycia

# C99 jest fajne!

## Lepsze tablice i struktury

### ► Przypisania do konkretnych pól

```
► struct tm my_time = {  
    .tm_hour = 21,  
    .tm_min = 37  
};
```

### ► Nienazwane lokalne struktury

```
► time_t t154 = mktime(&(struct tm) {  
    .tm_mday = 10,  
    .tm_mon = 3, // January == 0  
    .tm_year = 2010  
});
```

# C99 jest fajne!

## Typy danych (1/2)

- ▶ Jaki zakres ma `int`?
- ▶  $-2^{31} \dots 2^{31} - 1$  ?
- ▶ **ŹLE!**
  - ▶ Tak naprawdę - nie wiadomo (ಠ\_ಠ)
- ▶ Na szczęście - jest rozwiązanie!
  - ▶ `#include <stdint.h>`  
`#include <stdint.h>`
  - ▶ `int8_t`, `int16_t`, `int32_t`, `int64_t`
  - ▶ `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
  - ▶ `size_t` - typ wyniku zwracanego przez `sizeof`
  - ▶ `ptrdiff_t` - wynik odejmowania dwóch wskaźników
  - ▶ `(u)intptr_t` - `int` o rozmiarze równym rozmiarowi wskaźnika

# C99 jest fajne!

## Typy danych (2/2)

- ▶ Kiedy czego używać?
- ▶ Bierzmy pod uwagę to, skąd lub dokąd docierają dane
- ▶ Przykłady
  - ▶ Wartość przekazywana później do funkcji przyjmującej `int - int`
  - ▶ Rozmiar tablicy lub indeks do niej - `size_t`
  - ▶ Wartości dla protokołów sieciowych - `(u)int{8,16,32,64}_t`
- ▶ `printf, scanf`
  - ▶

```
#include <inttypes.h>
int16_t val;
scanf("%i" SCNd16, &val);
printf("Hello, %i" PRId16 "!\\n", val);
```
- ▶ Uwaga na overflow!
  - ▶ Według standardu wynik przekroczenia zakresu typów ze znakiem jest **NIEOKREŚLONY**.
  - ▶ Na niektórych architekturach powoduje przerwanie programu.



# C nie jest wyjątkowe!

- ▶ W C da się pisać:
  - ▶ obiektowo
  - ▶ komponentowo
  - ▶ funkcyjnie (sort of)
  - ▶ ...co nie znaczy że zawsze należy ;)
- ▶ W C można (**i należy!**) używać wzorców projektowych

# Jak ogarnąć moduły?



- ▶ W C nie ma namespace'ów
- ▶ ...dlatego tworzymy własne!
- ▶ Przykład na naszym githubie: [https://github.com/AVSystem/avs\\_commons](https://github.com/AVSystem/avs_commons)
- ▶ Funkcje "publiczne" mają nazwę zaczynającą się od nazwy modułu
  - ▶ `avs_buffer_*`, `avs_list_*`, `avs_unit_*`
- ▶ Funkcje wewnętrzne modułu (udostępnione pomiędzy jednostkami translacji)
  - ▶ `_avs_buffer_*`, `_avs_list_*`, `_avs_unit_*`
  - ▶ Nie są zdefiniowane w publicznych nagłówkach

# Programowanie prawie-funkcyjne w C

- ▶ Jedna z zalet wynikających z niskopoziomowości C
- ▶ Każda funkcja znajduje się pod jakimś, dobrze określonym adresem w pamięci
- ▶ Ten adres zawsze można pobrać i przekazać dalej...

# Programowanie prawie-funkcyjne w C:

## Deklaracje

### ▶ Funkcja

- ▶ `int func(double arg);`

### ▶ Typ funkcyjny

- ▶ `typedef int func_t(double);`

- ▶ Specyficzne własności - nie można prawie nic z nim zrobić!

### ▶ Wskaźnik na funkcję

- ▶ `int (*func_ptr)(double);`

- ▶ `func_t *func_ptr;`

### ▶ Typ wskaźnika na funkcję:

- ▶ `typedef int (*func_ptr_t)(double);`

- ▶ `typedef func_t *func_ptr_t;`

# Programowanie prawie-funkcyjne w C:

## Wskaźniki na funkcje (1/2)

- ▶ `int (*f) (const char *) = puts; // lub &puts`  
`f("Hello, world!"); // lub (*f) (...)`
- ▶ Wydajność?
  - ▶ Niższa niż przy bezpośrednim wywołaniu funkcji - potrzeba dereferencji wskaźnika
  - ▶ Czy mamy alternatywę?
    - ▶ enum reprezentujący listę możliwych funkcji
    - ▶ wybór w switchu lub drabince ifów
    - ▶ efekt - podobna wydajność, a tracimy na elastyczności

# Programowanie prawie-funkcyjne w C:

## Wskaźniki na funkcje (2/2)

- ▶ Wskaźnik na dowolne dane można rzutować na `void *`
- ▶ Wskaźnika na funkcję **nie można**.
- ▶ Dlaczego?
  - ▶ Harvard architecture - oddzielna pamięć na kod i dane
    - ▶ Stosowana w niektórych mikrokontrolerach
    - ▶ Adresy pamięci kodu i danych mogą mieć różny rozmiar!
  - ▶ Niektóre modele pamięci DOS - dalekie (segment+offset) vs. bliskie (offset) wskaźniki
- ▶ Jeżeli **BARDZO** potrzebujemy generycznego wskaźnika na dowolny kod, możemy rzutować na wskaźnik do dowolnej umownie przyjętej sygnatury
  - ▶ Np. `typedef int (*generic_fptr_t)(...);`
  - ▶ Koniecznie trzeba pilnować, żeby rzutować go do oryginalnego typu przed wywołaniem - inaczej nie wiadomo co się stanie...

# Programowanie prawie-funkcyjne w C:

## Praktyka (1/2)

### ► Rejestracja hooków użytkownika

```
► typedef int after_connection_t(...);  
int register_after_connection_hook(after_connection_t *hook);  
  
register_after_connection_hook(my_after_connection);
```

### ► Funkcje wyższego rzędu

```
► typedef int foreach_element_clb_t(element_t *element,  
                                void *user_data);  
  
int foreach_element(container_t *container,  
                  foreach_element_clb_t *clb,  
                  void *user_data);
```

► Zawsze dodawajmy wskaźnik na dane użytkownika do API!

# Programowanie prawie-funkcyjne w C:

## Praktyka (2/2)

```
typedef struct {
    size_t counter;
} count_elements_ctx_t;

static int count_elements_clb(element_t *element, void *ctx) {
    (void) element;
    ((count_elements_ctx_t *) ctx)->counter++;
    return 0;
}

size_t count_elements(container_t *container) {
    count_elements_ctx_t ctx = { 0 };
    foreach_element(container, count_elements_clb, &ctx);
    return ctx.counter;
}
```



# Programowanie obiektowe w C (1/4)

```
typedef struct animal_vtable animal_vtable_t;

typedef struct {
    const animal_vtable_t *vtable;
} animal_abstract_t;

struct animal_vtable {
    void (*make_sound)(animal_abstract_t *self);
    int (*eat)(animal_abstract_t *self, const char *meal);
};

void animal_make_sound(animal_abstract_t *self) {
    self->vtable->make_sound(self);
}

int animal_eat(animal_abstract_t *self, const char *meal) {
    return self->vtable->eat(self, meal);
}
```

# Programowanie obiektowe w C (2/4)

```
typedef struct {
    const animal_vtable_t *vtable;
    unsigned stomach_capacity;
} dog_t;

static void dog_make_sound(animal_abstract_t *self) {
    (void) self;
    printf("woof!\n");
}

static int dog_eat(animal_abstract_t *self_,
                  const char *meal) {
    dog_t *self = (dog_t *) self_;
    if (self->stomach_capacity >= strlen(meal)) {
        self->stomach_capacity -= strlen(meal);
        return 0;
    } else {
        return ERR_STOMACH_FULL;
    }
}
```

```
static const animal_vtable_t DOG_VTABLE = {
    .make_sound = dog_make_sound,
    .eat = dog_eat
};

animal_abstract_t *dog_new(void) {
    dog_t *dog = (dog_t *) malloc(sizeof(dog_t));
    if (!dog) {
        return NULL; // no memory
    }
    dog->vtable = &DOG_VTABLE;
    dog->stomach_capacity = 100;
    return (animal_abstract_t *) dog;
}
```

# Programowanie obiektowe w C (3/4)

```
typedef struct {
    const animal_vtable_t *vtable;
    bool good_mood;
} cat_t;

static int cat_make_sound(animal_abstract_t *self_) {
    cat_t *self = (cat_t *) self_;
    if (self->good_mood) {
        printf("meow!\n");
    } else {
        printf("purr >_\n");
    }
}

// ...
```

# Programowanie obiektowe w C (4/4)

```
int main(int argc, char *argv[]) {
    assert(argc == 3);
    animal_abstract_t *my_pet = NULL;
    if (strcmp(argv[1], "cat") == 0) {
        my_pet = cat_new();
    } else if (strcmp(argv[1], "dog") == 0) {
        my_pet = dog_new();
    }
    if (!my_pet) {
        printf("I can't have a %s\n", argv[1]);
        return -1;
    }
    animal_make_sound(my_pet);
    if (animal_eat(my_pet, argv[2])) {
        printf("My pet couldn't eat %s\n", argv[2]);
        return -1;
    }
    return 0;
}
```

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect.

# Dziękujemy!

Pytania?