

Przegląd właściwości C90/C99

Filozofia języka C

- Żadnych ukrytych kosztów związanych z wydajnością.
- Żadnych niejawnych mechanizmów związanych z konstrukcją kopiującą, przeciążeniem operatorów lub innych tego typu „ułatwiaczy”...
- C nie traktuje programisty jak dziecka któremu należy na każdym kroku zabraniać wykonywania operacji „niedozwolonych”. Jeśli kontekst programu nakazuje rzutować strukturę na float (zgodny z IEE754), to widocznie tego potrzebujesz (bo np. interfejs sprzętowy przyjmuje dane w tym formacie).
- Dokładnie zdajesz sobie sprawę z konsekwencji wykonywania nietypowych operacji bo wpływa na nie 3-5 czynników jasnych i łatwych do opanowania.
- Przenośność. Jeśli należy napisać kod właściwy dla danego sprzętu, możesz to zrobić. Masz także środki do tego by uczynić go przenośnym.

Wyjątek z opisu standardu:

Ufamy osobie programującej.

Nie bronimy przed wykonaniem czegoś czego co jest potrzebne w danej chwili w programie.

Utrzymujemy język mały i prosty.

Dostarczamy jedną drogę do wykonania danej operacji.

Program ma być szybki nawet kosztem przenośności.

Definicje zachowań „nieprecyzyjnych”

- **Zależne od implementacji** (ang. *implementation-defined*).
Dostawca kompilatora wybiera, implementuje i dokumentuje jedno z proponowanych rozwiązań
- **Niespecyfikowane** (ang. *unspecified*)
Jak poprzednie, jedynie nie wymaga dokumentowania podjętego wyboru.
- **Niezdefiniowane** (ang. *undefined*)
Wszystko się może zdarzyć. Włącznie z wizytą obcych spowodowaną działaniem kompilatora :-)
- **Specyficzne dla lokalizacji** (ang. *locale-specific*) - funkcje klasyfikacji liter, cyfr...

Zachowanie niezdefiniowane (ang. *Undefined Behavior*)

- Co to jest zachowanie niezdefiniowane?

Cokolwiek może się zdarzyć. Standard nie nakłada żadnych wymagań. Program może się nie skompilować lub może zostać wykonany niepoprawnie (powodując awarie lub generując niepoprawne wyniki) lub może przypadkowo wykonać dokładnie to, co zamierzał programista.

C FAQ

Zachowanie niezdefiniowane (ang. *Undefined Behavior*) (1/3)

- Powody „rozluźnienia” definicji języka:
 - Różnorodność obsługiwanych platform sprzętowych - na rynku występują architektury różne od von Neumanna
 - Nacisk na **ekstremalną wydajność** konstrukcji językowych przeniesionych do poziomu kodu maszyny - badania wykazują że brak kontroli zakresu pętli przyspiesza ją statystycznie o 30-50%.
 - Zachowanie prostoty składni i użycia samego języka - składnia i semantyka C, pozostaje dość jasna i przejrzysta.

Undefined Behavior - konsekwencje dla procesu wytwarzania (2/3)

- Niezbędny intensywny proces statycznego sprawdzania jakości kodu tak w trakcie kompilacji jak i narzędziami dedykowanymi.
- Niezbędny przestrzegany standard obejmujący zalecenia co do stosowania ryzykownych i nieporządkanych (generujących błędy) mechanizmów języka.
- Z racji wykonywana przeglądów kodu, zdefiniowanie sposobu zapisu kodu czytelnego i chroniącego przed błędami **i wymuszenie tego automatem a nie bezproduktywnymi dyskusjami.**
- Niezbędny intensywny proces testowania na platformie docelowej (oraz być może poza nią: symulatory/emulatory/maszyny wirtualne, środowiska referencyjne)

UB - konsekwencje dla procesu wytwarzania (3/3)

- Wymaganie stosowanie paradygmatów programowania zmniejszających prawdopodobieństwo popełnienia błędu.
- Niezbędne weryfikowanie także na poziomie wyniku kompilacji (binarne) powtarzalności operacji budowania kodu.
- Często uzasadnione implementowanie mechanizmów unikania i wychodzenia systemu z błędnych stanów.
- Zdefiniowanie procesu wytwarzania obejmującego wykładnie jakości wewnętrznej (np. dotrzymywane miary dla etapów CI oraz CD)

Obszary zachowań niezdefiniowanych (1/2)

- **Użycie niezainicjowanych zmiennych**

W świetle „egzotycznych” architektur, nie ma gwarancji przebywania tam **jakiegokolwiek wartości!**

- **Przepełnienie arytmetyki ze znakiem**

Standard nie definiuje sposobu przechowywania wartości ujemnych. **Nie należy przypuszczać (a tym bardziej bazować na tym) że jest to uzupełnienie do 2.**

- **Przesunięcia bitowe poza zakres słowa maszynowego**

Standard nie definiuje wyniku takiej operacji. Standaryzacja dla platform, wymagała by konieczności wykonania precyzyjnej operacji AND na wyniku a ekstremalna wydajność języka, jest w kolizji z tym rozwiązaniem.

Obszary zachowań niezdefiniowanych (2/2)

- **Dostęp do wskaźnika poza zakresem tablicy**

Standard gwarantuje poprawność arytmetyki wskaźników dla zakresu $[0, \text{max_index} + 1]$ i możliwość odczytu dla $[0, \text{max_index}]$. Usunięcie tej „niedogodności” zrujnowało by wydajność.

- **Dostęp do wskaźnika NULL**

Nie ma gwarancji by NULL miał wszystkie bity zgaszone. Standard wymaga **poprawnego porównania do 0**. Dostęp do pamięci wskazywanej na NULL może skończyć się ... czymkolwiek...

- **Łamanie zasad typowania**

Np. próby wyłuskania podwójnego wskaźnika przez pojedynczy. Pamięć docelowa może nie istnieć a platforma sprzętowa nie ma przewidzianej rozsądnej reakcji na wystąpienie takiego przypadku. Kompilator także zakłada wielkości typów danych stosując optymalizacje TBAA (ang. *Type-Based Alias Analysis*)

Zachowanie niezdefiniowane

- Standard C99 zawiera ponad 190 zachowań niezdefiniowanych.
- Standard C11, w dodatku J definiuje ich zakres zbliżony do 200.
- C nie był projektowany (patrz filozofia) jako „silnie ubezpieczający język maksymalnie bezpieczny dla radośnie-programujących”...

Zignorować to i nic z tym nie zrobić?

Standardy dobrych praktyk

- Problem UB jest jednym z zagadnień którymi zajmują się standardy branżowe oraz pokrewne:
 - MISRA - The Motor Industry Software Reliability Association C:2012 (MISRA C3)
 - AUTOSAR QA-C/C++ Compliance
 - CERT C, CERT C++ Compliance
 - ...
- Rozsądnie jest włączyć raporty narzędzi do procesu produkcji oprogramowania.

W dalszej części....




Kolejność operatorów

- Operatory w języku C nie zawsze wiążą tak jak (intuicyjnie) tego chcesz.
- **Operatory porównań wiążą silniej niż operatory bitowe.**
- **Operatory arytmetyczne wiążą silniej niż przesunięcia bitowych.**




```
if (a & 0xF0 > 0) ...
```

```
if ((a << SHIFT + 1) != 3) ...
```

Wiązanie operatorów (1/2)

Priorytet	Operator	Opis	Wiązanie
1	++ - () [] . -> (typ) { lista }	Post inkrementacja i dekrementacja Wywołanie funkcji Indeksowanie tablicy Dostęp do elementu struktury Dostęp do elementu struktury przez wskaźnik Literał dosłowny (ang. <i>compound literal</i>) (C99)	od lewej do prawej 
2	++ - + - ! ~ (typ) * & sizeof _Alignof	Pre inkrementacja i dekrementacja Unarny plus i minus Logiczne NOT i bitowe NOT Rzutowanie na typ Dereferencja/wyłuskanie Adres Wielkość elementu Wyrównanie (C11)	od prawej do lewej 
3	* / %	Mnożenie, dzielenie, modulo	od lewej do prawej 
4	+ -	Dodawanie i odejmowanie	
5	<< >>	Przesunięcia bitowe w lewo i prawo	
6	< <= > >=	Operacje relacji: mniejszy, mniejszy lub równy Operacje relacji: większy, większy lub równy	
7	== !=	Operacje relacji: równe, różne	
8	&	Bitowe AND	

Wiązanie operatorów (2/2)

Priorytet	Operator	Opis	Wiązanie
9	^	Bitowy XOR	od lewej do prawej 
10		Bitowy OR	
11	&&	Logiczny AND	
12		Logiczny OR	
13	?:	Operator trójargumentowy	od prawej do lewej 
14	= += -= *= /= %= <<= >>= &= ^= =	Przypisanie Przypisanie z sumą i różnicą Przypisanie z mnożeniem, dzieleniem i modulo Przypisanie z przesunięciem bitowym w lewo i prawo Przypisanie z operacją bitową AND, XOR, OR	
15	,	Przecinek	od lewej do prawej 

Wnioski:

- Operatory arytmetyczne mają wyższy priorytet niż logiczne
- „Operatory struktur” mają najwyższy priorytet
- Przesunięcia bitowe mają wyższy priorytet niż operacje logiczne i bitowe
- Operacje bitowe mają wyższy priorytet niż operacje logiczne
- od reguł są wyjątki i nieregularności...

Wiązanie operatorów - praktycznie

- **Nie zapamiętuj całej tablicy!**
- Normy branżowe jasno mówią że należy **jawnie wymuszać kolejność ewaluacji** z użyciem nawiasów „(” i „)” - BTW sprawdź jaki priorytet ma nawias...
- W przypadku referencji/dereferencji oraz jednoczesnej pre/post inkrementacji/dekrementacji, bądź wyrozumiały dla czytelnika i koniecznie postaw nawias.

```
/* Wiem że jesteś z tego dumny */  
while (*src++ = *dst++);
```
- ...

Wiązanie operatorów - przykłady niespodzianek

`r = m << 4 + 1 ... to r = m << (4 + 1) ...`

`*fun()... to TYP * fun(void)...`
jeśli chcesz wywołać funkcję to tak ...
`(*fun)() ...`

`*ptr++ ... to *(ptr)++ ... a nie (*ptr)++ ...`

`a & b == c & d ... to a & (b == c) & d ...`

`if (flags & FLAG != 0)... to if(flags & (FLAG != 0)) ...`

`a = b = c; to b = c; a = b;`

`while(c = rcve(i2c_bus) != END_COMMAND) ...`
to ...
`while(c = (rcve(i2c_bus) != END_COMMAND)) ...`

`int* a, b; to int * a; int b;`

L-wartość i R-wartość

- Operacje uzgadniania wartości i wiązania ich z nazwą (pojęcia z teorii typów), są obsługiwane w C z użyciem operatora przypisania.
- Wartości te zwyczajowo nazywane są:
 - L-wartość (ang. *L-value*) - wskazuje na pamięć która **może bezpiecznie pomieścić obiekt danego typu**, występuje po lewej znaku „=” i **można do niej przypisać wartość**
 - R-wartość (ang. *R-value*) - wskazuje na pamięć **gdzie obiekt o danej wartości występuje. Nie można do niego przypisać wartości**. Występuje po prawej znaku „=”

L-Wartość

- W języku C, L-Wartość to jedna z:
 1. Nazwa zmiennej jakiegoś typu (POD lub zdefiniowanego).
 2. Indeks specyfikowany z użyciem [i] nie będący nazwą żadnej tablicy.
 3. Wartość dereferencji nie wskazującej na żadną tablicę (*).
 4. L-Wartość w nawiasach.
 5. Obiekt opatrzony modyfikatorem dostępu const lub volatile (lub kombinacją poprzednich).
 6. Rezultat wyłuskania złożonego wskaźnika nie wskazujący na funkcję.
 7. Dostęp do wartości w strukturze/unii (-> lub .)

```
/* 1 */ int a;  
        int a = 42;  
        int b = a;  
  
/* 2 */ int * p = &a;  
        p[0] = 102;  
  
/* 3 */ *p = 103;  
  
/* 4 */ (a) = 323;  
  
/* 5 */ const double VAT = 0.23;  
  
/* 6 */ int * * r = &p;  
  
/* 7 */ struct { int a; double b; }  
        pr = {10, 3.14};  
        pr.b = 9.81;
```

R-Wartość - kilka przykładów (1/3)

```
#include <stdio.h>

int table[] = { 10, 20, 30};

int * calc(int a, int b) {
    if((a + b) % 2) {
        return table + 1;
    }
    return table + 2;
}

int main(void) {
    *(calc(10, 10)) += 1;
    printf("%d %d\n", table[1], table[2]); /* 20 31 */
    *(table + 1) = 200;
    0[table] = 1000;
    printf("%d %d %d\n", table[0], table[1], table[2]); /* 1000 200 31 */

    int k = 1;
    printf("k is odd? %c\n", "FT"[k % 2]); /* k is odd? T */

    return 0;
}
```

To nie znaczy że namawiam do tworzenia takiego kodu :)

R-Wartość - kilka przykładów (2/3)

- Przecież działa i jest ok... kompilator nie zgłasza ostrzeżeń... :-/

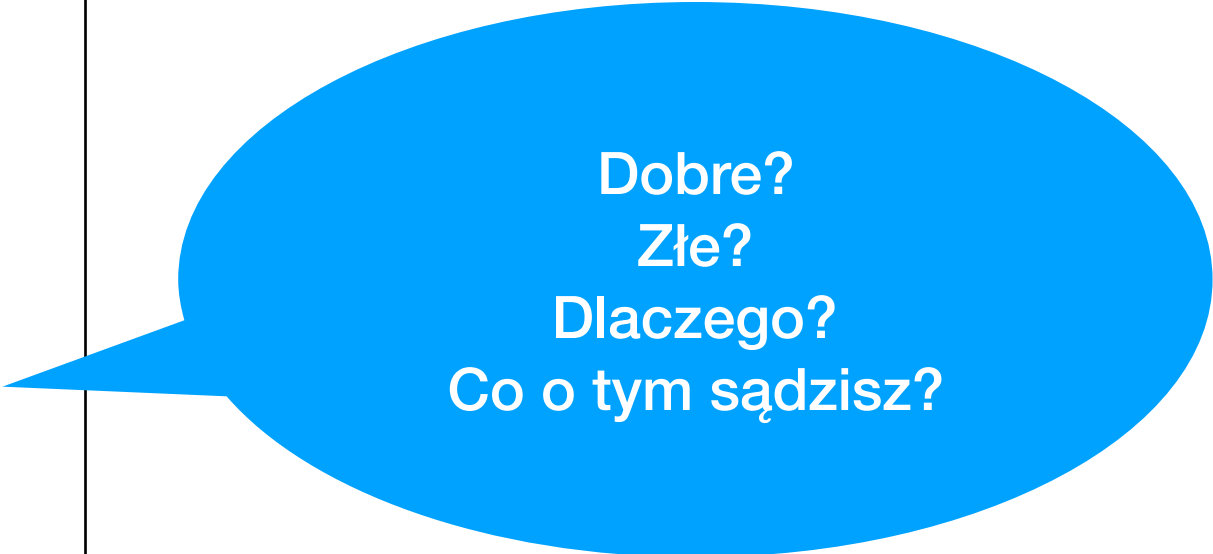
```
#include <stdio.h>

struct Msg {
    char payload[8];
};

struct Msg who(void) {
    struct Msg result = { "Lemon" };
    return result;
}

struct Msg name(void) {
    struct Msg result = { "John" };
    return result;
}

int main(void) {
    printf("%s %s.\n", name().payload, who().payload);
    return 0;
}
```



Dobre?
Złe?
Dlaczego?
Co o tym sądzisz?

R-Wartość - kilka przykładów (3/3)

- Chyba lepiej... (nieco)...

```
/* ... */  
  
int main(void) {  
    struct Msg my_name = name();  
    struct Msg my_who = who();  
    printf("%s %s.\n", my_name.payload, my_who.payload);  
    return 0;  
}
```

- Można jeszcze lepiej/bezpieczniej?

Proste (acz niekompletne) środki obrony...

- Dla kompilatorów z rodziny gcc lub clang:
 - -Wcast-*
 - -Wuninitialized
 - -fsanitize-*=
 - -fsanitize=?? - undefined, shift, integer-divide-by-zero, signed-integer-overflow, ...
 - ...
- Dodatkowo:
 - clang static analyzer - scan-build
 - Valgrind memcheck
 - Klee LLVM Execution Engine
 - c-semantic-tools
 - ...

Ćwiczenie - zrobmy to razem

- Czy ten kod się kompiluje?
- Czy ten kod działa? Jeśli tak to jak?
- Co można w nim poprawić by ... „się nie wstydzić”? :)

```
main() {  
    char * msg = "Hello C!\n";  
    printf(msg);  
}
```