Język C

Kierunek: Informatyka Stosowana (*Wydział FAIS UJ*)

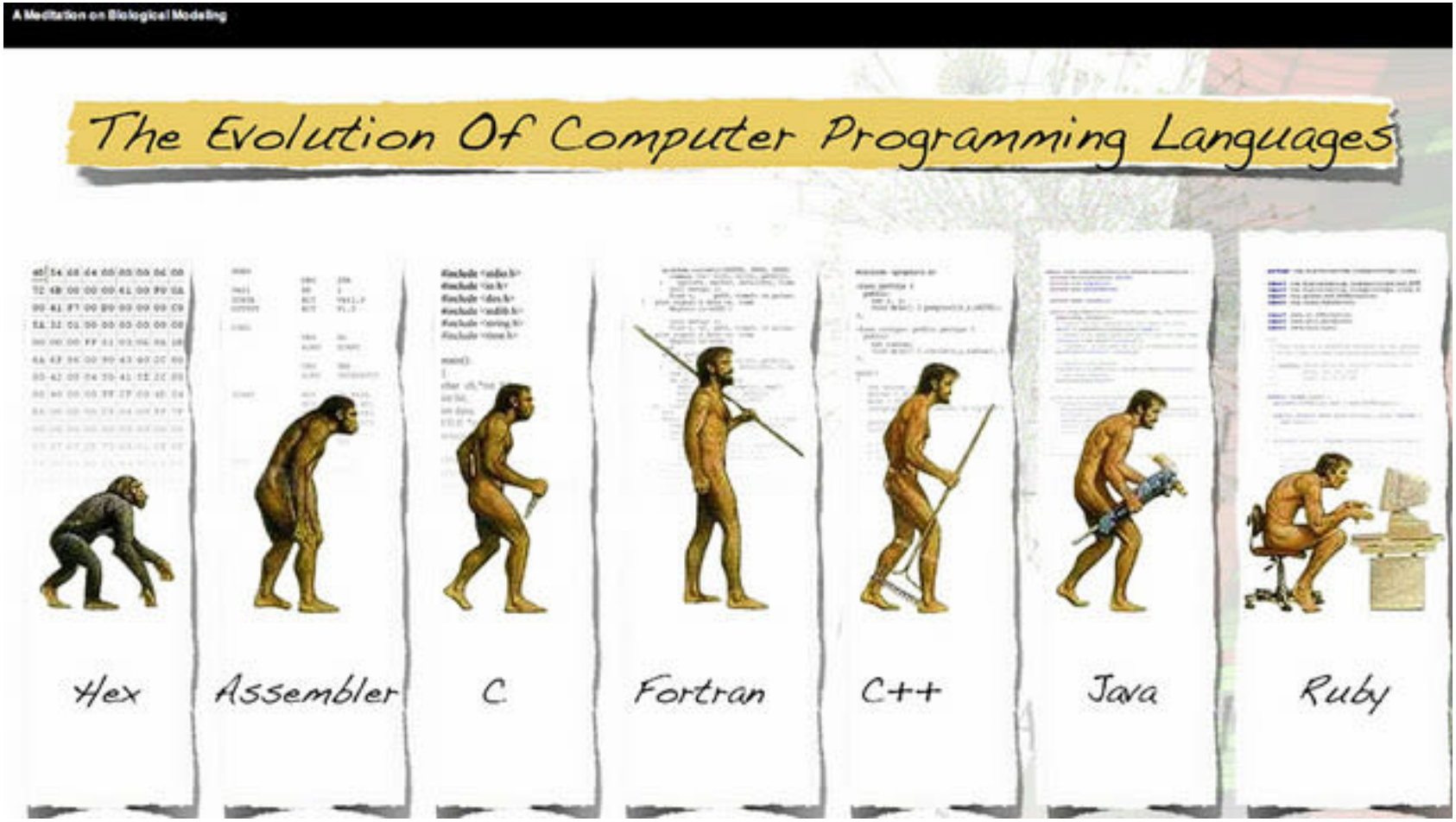
Wykład: Wtorek, 12.15 s. A-1-06

Forma zaliczenia kursu: Egzamin pisemny (test wyboru) * **

* Warunkiem przystąpienia do egzaminu jest wcześniejsze ***zaliczenie ćwiczeń*** (lub w uzasadnionych przypadkach: *zgoda prowadzącego ćwiczenia*)

** **Ocena 5.0 (bdb)** z ćwiczeń *zwalnia z pisemnej części egzaminu*

Po co uczyć się języka C?



https://lifehacker.com/the-case-for-learning-c-as-your-first-programming-langu-1682070792

How to make LISP go faster than C

Didier Verna*

Abstract

Contrary to popular belief, LISP code can be very efficient today: it can run as fast as equivalent C code or even faster in some cases. In this paper, we explain how to tune LISP code for performance by introducing the proper type declarations, using the appropriate data structures and compiler information. We also explain how efficiency is achieved by the compilers. These techniques are applied to simple image processing algorithms in order to demonstrate the announced performance on pixel access and arithmetic operations in both languages.

statically (hence known at compile-time), just as you would do in C.

Safety Levels While dynamically typed LISP code leads to dynamic type checking at run-time, it is possible to instruct the compilers to bypass all safety checks in order to get optimum performance.

Data Structures While LISP is mainly known for its basement on list processing, the COMMON-LISP standard features very efficient data types such as specialized arrays, structs or hash tables, making lists almost completely obsolete.

http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_19.pdf

Poprzedni wykład: *krótkie przypomnienie*

- Komputer to maszyna cyfrowa *(=> zasady programisty)*
- Reprezentacja liczb w maszynie *(będzie ERRATA …)*
- Krótka charakterystyka języka C
- *Jednostki leksykalne języka C*

ERRATA: typy liczbowe w języku C:

Całkowite: `char` — 1 bajt/8 bitów,

zakres: $0 \div 2^8\text{-}1=255$ (jeśli *unsigned***; *signed:* $-128 \div 127$)

`int` — ***co najmniej**** 2 bajty/16 bitów; zakres:
$0 \div 2^{16}\text{-}1=65635$ (*unsigned*) | $-32768 \div 32767$ (*signed*)

*) *Zwykle typ int odzwierciedla romiar l.całkowitej wynikający z architektury maszyny, współcześnie: 32 lub 64 bajty.*

**)*Przykładowa pełna deklaracja zmiennej:* `unsigned int i;`
***stałe** zapisujemy np.* `77` *(jeśli int),* `77u` *lub* `77U` *(jeśli unsigned)*

UWAGI o typie *char*:

- Standard ANSI C nie określa, czy char ma znak czy nie.
- Jest jednak pewne, że znaki *klawiaturowe* będą reprezentowane przez liczby **nieujemne**
- Jeśli nasza zmienna typu char ma przechowywać coś *innego niż znaki*, deklarujemy ją **jawnie** jako `signed char` ALBO jako `unsigned char`. (*Albo po prostu jako* `int` *...*)

Przyporządkowanie wartości całkowitych znakom daje ciekawe możliwości, np. instrukcja:

```
if ( s>='0' && s<='9' )   d = s - '0';
```

przetwarza cyfrę/znak (`char s`) na cyfrę/liczbę (`int d`).

(Warunek powyżej odpowiada: `isdigit(s)` *z biblioteki* <**ctype.h**> *...)*.

Typy zmiennopozycyjne:

`float` — pojedynczej precyzji, 32 bity (*co najmniej!*), ~7 cyfr, zakres:

$-3.4 \times 10^{38} \div -3.4 \times 10^{-38}$, 0, $3.4 \times 10^{-38} \div 3.4 \times 10^{38}$

double — podwójnej precyzji, 64 bity (*co najmniej!*), ~15 cyfr, zakres:

$-1.7 \times 10^{308} \div -1.7 \times 10^{-308}$, 0, $1.7 \times 10^{-308} \div 1.7 \times 10^{308}$

(*Maksymalne/minimalne wartości zdefiniowane w* `<limits.h>`
Poszczególne implementacje często uzupełniają typy zmiennopozycyjne o użyteczne stałe, np. `Inf` `-Inf` `NaN` *- "not a number"*)

Kwalifikatory `short` (krótki) oraz `long` (długi) modyfukują typy podstawowe, np. deklaracja: `long int i;` wymusza 32 bity.

(W tym przypadku słowo `int` ***może zostać pominięte;*** *ale np.* `long double` *już nie można skrócić …)*

6 jednostek leksykalnych języka C

1) słowa kluczowe: `int, goto, if, else, …`
2) stałe: `0xF85  2.17e+10  ‘a’  ‘7’`
3) łańcuchy znaków: `“Tekst”` <=> tablicy:

   `{’T’,’e’,’k’,’s’,’t’,’\0’`
4) identyfikatory: `A-Z, a-z, 0-9`

   (pierwszy znak: zawsze litera lub ‘_’)
5) operatory: `=  +  *  +=  &&  ,  ?:`
6) separatory: `‘ ‘  ( )  [ ]  { }  ;`

Stałe

- W języku stałe mają swoje typy;
- Typu stałej *nie deklarujemy jawnie*, **jest dedukowany przed kompilator** na podstawie **samego zapisu (!)**
  (`123` - *int*, `123l` lub `123L` - *long*, `123ul` lub `123UL` - *unsigned long*, `7.0` - *double*, `7.0f` lub `7.0F` - *float*, `7.0l` lub `7.0L` - *long double*)
- Stałe napisowe (`"Napis"`) mają specjalne prawa, nie można ich np. porównać ze sobą operatorami: `==` lub `!=`

=> Zapis `'7'` oznacza stałą znakową (char), zaś zapis `"7"` stałą napisową, równoważną tablicy znaków `{'7','\0'}`.
*(Element `'\0'` to tzw. **znak pusty**, oznaczający koniec napisu.)*

PRZYKŁAD: Obliczamy długość napisu w tablicy s

```
int strlen(char s[])
/*  Może być też:  “strlen(const char s[])” (!) */
{
   int i;

   i=0;
   while (s[i]!=‘\0’)
      i++;
   return i;
}
```

Znaki specjalne w C

\a	znak alarmu	\\	znak \
\b	znak cofania	\?	znak zapytania
\f	znak nowej strony	\'	znak apostrofu
\n	znak nowego wiersza	\"	znak cudzysłowu
\r	znak powrotu karetki	*ooo*	liczba ósemkowa
\t	znak tabulacji poziomej	\x*hh*	liczba szesnastkowa
\v	znak tabulacji pionowej		

Ale już np. **znak końca pliku** (ang. *end-of-file*) `EOF` jest zdefiniowany jako stała (typu `int`) w nagłówku `<stdio.h>`.

(Pisząc na standardowe wejście, można wprowadzić EOF używając kombinacji klawiszy `CTRL-D`*.)*

Wyrażenia stałe

Wartości wyrażeń stałych (tj. wyrażeń, *w których występują wyłącznie stałe*) są obliczane na etapie kompilacji programu, wynik trafia do kodu maszynowego. Np. w deklaracji tablicy:

```
double x[1977-25+1];
/* pomiędzy nawiasami [ ] mamy wyr. stałe */
```

Ciekawą możliwością jest **dodawanie napisów**. Deklaracja:

```
char TSEliot[]=
   "Pomiędzy ideę\n" "I rzeczywistość\n"
   "Pomiędzy zamiar\n" "I dokonanie\n" "Pada Cień\n";
```

poprawnie zainicjuje tablicę znakową.

Makra czy kwalifikator *const* ? (cz.1)

Stałe w C to elementy jak: `123` `123UL` itp. Makrodefinicje:

```
#define NMAX 100
#define ALPH 0.7L
```

to nie "definicje stałych" a raczej definicje nazw/symboli (NMAX oraz ALPH), które kompilator podmieni na stałe (100 oraz 0.7L).

Z kolei deklaracja:

```
const int nmax = 100;
```

nie definiuje stałej, lecz raczej *"zmienną, której wartość nie może ulec zmianie"*. (Ma ona np. swój adres, dostępny jako &nmax).

(Zwyczajowo, makra nazywamy używając wielkich liter, zmienne - małych.)

Makra czy kwalifikator *const* ? (cz.2)

A czy można zdefiniować:

```
#define NMAX1 NMAX+1
```

Jak najbardziej, ale wówczas np. wyrażenie `2*NMAX1` zostanie zastąpione przez `2*100+1` [`==201`], *a "poeta" prawdopodobnie miał na myśli*: `2*(100+1)` [`==202`] (**!**).

Jak uzyskać porządany efekt? Stosuj nawiasy (!), tutaj:

```
#define NMAX1 (NMAX+1)
```

"Za dużo nawiasów nie przeszkadza, za mało nawiasów - nie pomaga".

Inny sposób: `const int nmax1=nmax+1;`

… i już ból głowy przechodzi …

Makra czy kwalifikator *const* ? (cz.3)

Podsumowanie:

- Makra (#define) to *potężne narzędzie*, jednak ich niefortunne definicje często prowadzą do *trudnych do wykrycia błędów*.
  (*=> nawiasy pomagają !!!*)
- Makrami można niekiedy **zastąpić fukcje**, np.:

```
#define SQR(x) ((x)*(x))

#define MAX(a,b) (((a)>(b))?(a):(b))
```

- W przypadku użycia słowa kluczowego const do tzw. *"deklaracji stałych"*, czas wykonania programu może być nieco dłuższy (odwołanie do adresu, odczyt komórki pamięci), jednak w praktyce **dzisiaj już nie warto się tym przejmować** …
  (*=> niech się komputer męczy, nie ja !*)

Typ i stałe wyliczeniowe (enum)

Skończony ciąg stałych można zdefiniować deklaracją:

```
enum months { JAN=1, FEB, MAR, APR, MAY, JUN,
   JUL, AUG, SEP, OCT, NOV, DEC }
```

Wówczas, mamy: JAN==1, FEB==2, …, DEC==12.
Wartości w jednym wyliczeniu mogą się powtarzać; jeśli nie zdefiniujemy pierwszej - domyślnie przyjmowane jest 0.
Dodatkowo, pojawia się **pełnoprawny typ danych**: `enum months`.

```
#include<stdio.h>

enum week{Mon=1, Tue, Wed, Thu, Fri, Sat, Sun};

int main()
{
   enum week day;     /* To zmienna typu “enum week” */
   day = Wed;
   printf(“%d\n",day);     /* Wypisuje: “3” */
   return 0;
}
```

[*https://www.geeksforgeeks.org/enumeration-enum-c/*, zmiany: AR]

Te same stałe całkowite można zdefinować też np. tak:

```
enum week{Sat=6, Sun, Mon=1, Tue, Wed, Thu, Fri};
```

wówczas także dostajemy Mon=1, Tue=2, … , Sun=7 (**!**)

Uwaga praktyczna: W programach *kalendarzowych* często zachodzie potrzeba drukowania / odczytu nazw dni tygodnia lub miesięcy jako napisów (“Jan”, “Tue” itp.) przy zachowaniu ich logicznego związku (*przyporządkowania*) z odp. liczbami.

W takim wypadku - używamy **struktur** (*i tablic struktur* …)

(*Zmienne typu wyliczeniowego - raczej żadko używane.*)

Deklaracje

Co do zasady, w C każda zmienna **musi być zadeklarowana** przed pierwszym użyciem [*często jednak możliwa* ***deklaracja niejawna*** *– „przez kontekst”*].

Przykładowo, deklaracje:

```
int i,j,k;
```

oraz

```
int i;
int j;
int k;
```

są **równoważne.** *=> A po co komu drugi sposób?*

>> **Ponieważ zostaje więcej miejsca na komentarze!** <<

W deklaracjach można od razu nadawać zmiennym **wartości początkowe,** np:

```
double eps=1.0e-6;
```

po prawej stronie operatora przypisania (=) musi znajdować się (*co do zasady…*) **stała** lub **wyrażenie stałe** (*już dzisiaj było …*).

Ważny wyjątek stanowią tzw. **zmienne automatyczne**, tj. *zmienne lokalne funkcji,* które pojawiają się i znikają za każdym wywołaniem funkcji. Takie zmienne można inicjować ***dowolnym wyrażeniem*** (np: `int i=a+b;` `float z=sqrt(7.0f);` itp.).

Jeśli zmienna automatyczna niezainicjowana - przyjmuje wartość przypadkową (potocznie: *"przechowuje śmieci"*).

Zasięg zmiennych:

Zmienne lokalne widoczne są wewnątrz bloku ({ … }), w którym zostały zadeklarowane (także w blokach w nim zagnieżdżonych).

Zmienne zewnętrzne (deklarowane poza wszystkimi blokami) - ***istnieją cały czas;*** jeśli niezainicjowane - *podstawiane zera*.

Deklaracja (a właściwie: *definicja*) zmiennej lokalnej ***zakrywa*** zmienną zewnętrzną (lub lokalną z bloku wyżej) o tej samej nazwie.

„Każdą zmienną w języku C można zadeklarować wielokrotnie, ale tylko raz zdefiniować.”

CO TO ZNACZY ????

Kwalifikator *extern*: Przykład

```
#include <stdio.h>

int j=1;  /* Definicja zmiennej zewn. */

int main()
{
   extern int j;  /*Ponowna deklaracja, tutaj*/
        /* nie wolno przypisać wartości (!) */

   printf("%d\n",j);   /* wypisuje: "1" */
   return 0;
}
```

A jeśli pominiemy *extern* ?

```
#include <stdio.h>

int j=1;  /* Definicja zmiennej zewn. */

int main()
{
   printf("%d\n",j);   /* TEŻ DOBRZE ! */
   return 0;
}
```

W tym przykładzie mamy tzw. **deklaracje niejawną** (*przez kontekst*)

A gdyby tak nigdy nie używać *extern* ?

```
#include <stdio.h>

int j=1;  /* Definicja zmiennej zewn. */
/* … TUTAJ DUZO INNYCH DEFINICJI … */

int main()
{
   int j;   /* … przypadkowo się-napisało … */
   /* … */
   printf("%d\n",j);   /* wypisuje "śmieci"! */
   return 0;
}
```

- ***Zmiennych zewnętrznych nie należy nadużywać!***
- *Przydają się* np. wtedy, gdy wiele funkcji programu działa na *tych samych* danych (*a listy parametrów funkcji rosną i rosną …*).
- Najlepiej nadawać im ***unikalne*** (tj. ***długie!***) nazwy, aby przypadkowo nie zostały zakryte przez *zmienne* lokalne.
- W dłuższych programach - zmienne zewnętrzne warto **redeklarować** z kwalifikatorem **extern** (wówczas deklaracja innej zmiennej lokalnej o t.samej nazwie *spowoduje błąd kompilacji …*)

Kwalifikator *static*:

Zmienna lokalna poprzedzona **static** zachowuje się (*prawie …*) jak zewnętrzna, tzn. pamięta wartość pomiędzy wywołaniami funkcji, (*choć oczywiście nie jest widoczna poza swoim blokiem* { … }).

Jeśli *zmienna zewnętrzna* jest zadeklarowana jako **static**, będzie widoczna ***wyłącznie w "swoim" PLIKU*** (ale można ją *redeklarować* w innym pliku za pomocą **extern**).

ŁĄCZNIE:

- ***W małym programie*** (zwłaszcza *jednoplikowym*) w zasadzie możemy zapomnieć o kwalifikatorach **extern** i **static**: kiedy potrzeba, aby zmienna *pamiętała swoją wartość*, definiujemy ją jako zewnętrzną (tzn. ***poza wszystkimi blokami***), a później używamy *przez kontekst* (*pamiętając, aby nie zakryć nazwy!*)
  W pozostałych przypadkach - używamy **zmiennych automatycznych**.
- **A kiedy *„mały program"* urośnie … => extern + static**

Przykład podsumowujący *elementarz zmiennych*:

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

int j=2;
/* double x=sqrt(1.0*j);  >> TUTAJ NIE WOLNO! << */

int main()
{
   extern int j;     /* Tutaj nie piszemy "j=2" */
   int k;
   double x=sqrt(1.0*j);  /* >> A TUTAJ JUŻ WOLNO << */

   printf("%d\n",j);    /* Wypisuje "2"      */
   printf("%d\n",k);    /* Wypisuje śmieci   */
   printf("%f\n",x);    /* Wypisuje "1.414214" */
   return 0;
}
```

Operatory - proste przykłady

Przypisanie (*podstawienie*), ***dodawanie*** i ***mnożenie***:

```
int a,b,c;   /* … oraz operator przecinkowy */

a=8;
a=a+1;   /*  Wersja skrócona:  a+=1;  */
a=a*a;   /*  <=>  a*=a;  */

/* … */
c=2*(b=a=a+7);   /* Czytamy OD PRAWEJ! */
```

Priorytety i łączność operatorów w C

Operatory	Łączność
() [] -> .	lewostronna
! ~ ++ -- + - * & (*typ*) sizeof	prawostronna
* / %	lewostronna
+ -	lewostronna
<< >>	lewostronna
< <= > >=	lewostronna
== !=	lewostronna
&	lewostronna
^	lewostronna
|	lewostronna
&&	lewostronna
||	lewostronna
?:	prawostronna
= += -= *= /= %= ^= |= <<= >>=	prawostronna
,	lewostronna

Jednoargumentowe operatory +, -, * oraz & mają priorytet wyższy niż ich odpowiedniki dwuargumentowe.

Operatory arytmetyczne

W C mamy ***dwuargumentowe*** operatory *arytmetyczne*:

```
+   -   *   /
```

oraz **dzielenie modulo** (%) – wyłącznie dla *typów całkowitych*.

Na przykład, wyrażenie:

```
(( y%4 == 0 && y%100 != 0 ) || y%400 == 0)
```

przyjmuje wartość 1 jeśli y to rok przestępny (w kalendarzu *gregoriańskim*) albo 0 jeśli nie.

Dla ***liczb ujemnych*** - wynik dzielenia modulo (%) może zależeć od maszyny; podobnie jest z dzieleniem całkowitym (/).

Relacje i operatory logiczne

Operatory relacji: `>` `>=` `<` `<=`

Operatory porównania: `==` `!=`

Wszystkie mają niższy priorytet niż operatory arytmetyczne,
a zatem wyrażenia: `i<lim-1` oraz `i<(lim-1)` **są równoważne**.

Operatory logiczne to: `&&` `||` (*„i” „lub”*); oraz *negacja*: `!`

Wyrażenia zawierające operatory logiczne obliczane są od lewej do prawej, ale *tylko do momentu, gdy wynik jest już jednoznaczny* (!)

```
for (i=0; i<lim-1&&(c=getchar())!='\n'; i++)  s[i]=c;
```

=> dla `i==lim-1` znak ***już nie będzie wczytany***.

Interpretacje *logiczna* liczb całkowitych

Jeśli x jest typu int, warunek if(x) będzie interpretowany jako:

PRAWDZIWY, dla dowolnego x != 0

FAŁSZYWY, wyłącznie dla x == 0.

Jeśli z kolei zmiennej całkowitej przypisujemy wartość wyrażenia logicznego, np. `x = (a>b);` wynik będzie równy 1 (*zawsze* 1), jeśli wyrażenie po prawej stronie okazało się *PRAWDZIWE*, albo 0, jeśli okazało się *FAŁSZYWE*.

Konsekwentnie, **logiczna negacja** (!x) daje 0 dla dowolnego x != 0, albo 1 wyłącznie dla x == 0.

Np. warunek na *rok przestępny* można przepisać np. tak:

```
if(( !(y%4) && y%100 ) || !(y%400))
```

Przekształcenia typów

Jeśli argumenty operatora są *różnych typów*, dokonuje się przekształcenia typu *„ciaśniejszego”* do *„obszerniejszego”* (*ale tylko w ramach jednego wyrażenie składowego …*)

Na przykład, dodawanie: `1.0f + 1` da wynik float, ale już podstawienie: `y = (1/3)*x*x;` gdzie x - float (lub double) okazuje się równoważne `y=0.0f` (lub `y=0.0`) … Dzieje się tak, ponieważ `(1/3)` jest *najpierw* obliczane jako int, a dopiero *później* przekształcane na float (lub double). *A wystarczyło napisać:* 1.0/3 *…*

Zaskakiwać mogą też *porównania z argumentami różnych typów*, np. ***prawdziwe*** okazuje się:

`-1L < 1U` jak również: `-1L > 1UL` (**!!!**)