

```

char
_3141592654[3141
], __3141[3141]; _314159[31415], _3141[31415]; main(){register char*
_3_141,*_3_1415, *_3__1415; register int _314, _31415, __31415,*_31,
_3_14159, __3_1415;*_3141592654=_31415=2, _3141592654[0][_3141592654
-1]=1[_3141]=5; _3_1415=1;do{ _3_14159=_314=0, _31415++;for( _31415
=0;_31415<(3,14-4)*__31415;_31415++)_31415[_3141]=_314159[_31415]= -
1;_3141[*_314159=_3_14159]=_314;_3_141=_3141592654+__3_1415;_3_1415=
__3_1415 +__3141;for
_3_1415 ;
_3_141 ++,
+=_314<<2;
*_3_1415;_31
if(!(*_31+1)
_31415,_314
_31415 ;*(
)+=*_3_1415
_3_1415 >=
_3_1415+= -
)++;_314=_314
_3_14159 && *
=1, __3_1415 =
_314+(__31415
while ( ++ *
)*_3_141--=0
); { char *
write((3,1),
),(_3_14159
3.1415926; }
_31415<3141-
31415% 314-(
_31415
] +
[ 3]+1)-_314;
_3141592654))

```

Program w C

ver. 12 z drobnymi modyfikacjami!

Wojciech Myszk

Katedra Mechaniki, Inżynierii Materiałowej i Biomedycznej

2021-08-10 08:42:46 +0100

```

_3141592654[3141
_31415;_31415
_3_1415++){_314
_314<<-1;_314+=
_314159;_314;
)*_31=_314 /
[_3141]=_314 %
_3_1415=_3_141
=*_31;while(*
31415926;)*
10,(*_3_1415
[_3141]; if ( !
_3_1415)_3_14159
_3_14159
>>1)>=_31415 )
_3_141==3141/314
; }while( _3_14159
_3_1415;
(--*_3_14, _3_14
++,++_3_14159))+
for ( _31415 = 1;
1;_31415++)write(
3,14),_3141592654[
"0123456789", "314"
puts((*_3141592654=0
;_314= *"3.141592");

```



Tak wygląda program w języku C

```
1 /* Hello World in C, Ansi-style */  
2 #include <stdio.h>  
3 int main(void)  
4 {  
5     puts("Hello _World!");  
6 }
```



A tak (ten sam) w Pascalu

```
program test;  
{rozne deklaracje}  
begin  
    writeln ( 'to _jest _program' );  
end.
```



Program składa się z:

1. Pewnej **struktury**;



Program składa się z:

1. Pewnej **struktury**;
2. **Poleceń** wykonywanych przez procesor;



Program składa się z:

1. Pewnej **struktury**;
2. **Poleceń** wykonywanych przez procesor;
3. Obiektów (zwanymi **zmiennymi**) służących do przechowywania danych, wyników i wartości pośrednich uzyskanych podczas obliczeń;



Program składa się z:

1. Pewnej **struktury**;
2. **Poleczeń** wykonywanych przez procesor;
3. Obiektów (zwanymi **zmiennymi**) służących do przechowywania danych, wyników i wartości pośrednich uzyskanych podczas obliczeń;
4. **Stałych** używanych podczas obliczeń i do inicjowania wartości zmiennych;



Program

```
/*  
    Hello World in C, Ansi-style  
*/  
#include <stdio.h>  
int main(void)  
{  
    int z;  
    z = z + 1;  
    puts("Hello World!"); // druk  
}
```



Program

► komentarz

```
/*  
Hello World in C, Ansi-style  
*/  
#include <stdio.h>  
int main(void)  
{  
    int z;  
    z = z + 1;  
    puts("Hello World!"); // druk  
}
```



Program

► struktura

```
/*  
    Hello World in C, Ansi-style  
*/  
#include <stdio.h>  
int main(void)  
{  
    int z;  
    z = z + 1;  
    puts("Hello World!"); // druk  
}
```



Program

- ▶ struktura
 - ▶ deklaracje

```
/*  
    Hello World in C, Ansi-style  
*/  
#include <stdio.h>  
int main(void)  
{  
    int z;  
    z = z + 1;  
    puts("Hello World!"); // druk  
}
```



Program

► polecenia

```
/*  
    Hello World in C, Ansi-style  
*/  
#include <stdio.h>  
int main(void)  
{  
    int z;  
    z = z + 1;  
    puts("Hello World!"); // druk  
}
```



Program

- ▶ identyfikatory (zmienne?)

```
/*  
    Hello World in C, Ansi-style  
*/  
#include <stdio.h>  
int main(void)  
{  
    int z;  
    z = z + 1;  
    puts("Hello World!"); // druk  
}
```



Program

► stałe

```
/*  
    Hello World in C, Ansi-style  
*/  
#include <stdio.h>  
int main(void)  
{  
    int z;  
    z = z + 1;  
    puts("Hello World!"); // druk  
}
```



Program

► zmienne

```
/*  
    Hello World in C, Ansi-style  
*/  
#include <stdio.h>  
int main(void)  
{  
    int z;  
    z = z + 1;  
    puts("Hello World!"); // druk  
}
```



Program

```
/*  
    Hello World in C, Ansi-style  
*/  
#include <stdio.h>  
int main(void)  
{  
    int z;  
    z = z + 1;  
    puts("Hello World!"); // druk  
}
```

► funkcje

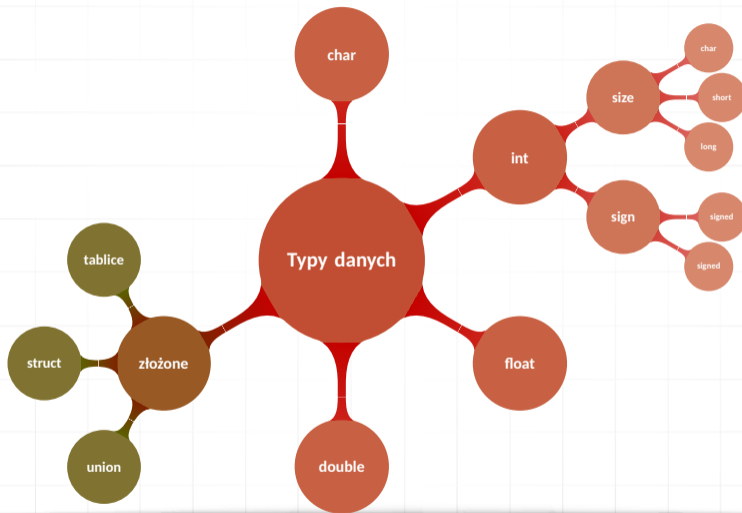


Zmienne

1. Każdy program operuje na pewnych obiektach służących do przechowywania bieżących wartości.
2. Obiekty te nazywa się **zmiennymi**.
3. Każda zmienna musi mieć jakąś nazwę. **Pierwszym znakiem nazwy powinna być litera**, używać można również cyfr i znaków podkreślenia.
4. W odróżnieniu od różnych innych pojemników — te do przechowywania wartości rozróżniają typ przechowywanej wartości.
5. Jak zmienna przechowuje jedną wartość — nazywa się **zmienną prostą**, gdy potrafi przechować więcej wartości — **złożoną**.



Typy danych



Reprezentacja binarna: teksty

1. Każdy znak tekstu to jeden bajt.
2. Zmienna tych **char** zajmuje jeden bajt.
3. Reprezentacją binarną znaku ASCII jest liczba całkowita (bez znaku) o wartości równej **kodowi ASCII** tego znaku.



Reprezentacja binarna: liczby całkowite

1. Każda liczba całkowita tekstu to jeden, dwa, cztery lub osiem bajtów...
2. Zmienna typu **int** zajmuje cztery bajty (czyli 32 bity).
3. Liczby całkowite (ze znakiem) reprezentowane są w **zapisie uzupełnieniowym do dwu**.
4. Najbardziej znaczący bit to bit znaku (1 oznacza liczbę ujemną, 0 dodatnią)
5. Liczby całkowite bez znaku reprezentowane są w kodzie binarnym.



Reprezentacja binarna: liczby zmiennoprzecinkowe

1. Każda liczba zmiennoprzecinkowa zajmuje 4, 8 lub 16 bajtów.
2. Dokładnie sposób zapisu liczb definiuje norma IEEE 754.
3. Stosuje się zapis wykładniczy w postaci $\text{znak} * \text{mantysa} * 2^{\text{cecha}}$.
4. Znak to jeden bit, cecha (wykładnik) to 8, 11 albo 15 bitów (binarna liczba całkowita ze znakiem); mantysa (ułamek binarny) — pozostałe bity.
5. Liczby zapisywane są w postaci znormalizowanej (przed kropką dziesiętną musi być 1 — cyfra różna od zera).
6. Specjalne „wartości” $+\infty$, $-\infty$ oraz NaN (Not a Number) mają również swoje reprezentacje binarne; takie wyniki mogą pojawić się w przypadku „dzielenia przez zero”.



Nazwy zmiennych

1. Każdy identyfikator musi być unikatowy!
2. Wielkie i małe litery są ważne.
3. Nazwy identyfikatorów powinny zaczynać się od litery i składać z liter i cyfr.
4. Znak _ (podkreślenie) traktowany jest jak litera i może służyć do tworzenia bardziej czytelnych nazw.
5. Znak _ nie powinno się stosować na początku nazwy — bardzo wiele nazw systemowych zaczyna się od niego.
6. Nie należy używać polskich liter w nazwach.
7. Tradycyjnie zmienne pisze się małymi literami, a różnego rodzaju stałe — wielkimi.
8. Słowa kluczowe (**int**, **void**, **if**, **else**,...) są zarezerwowane i nie mogą być używane jako nazwy.



Słowa kluczowe języka C

auto

const

double

float

int

short

struct

unsigned

break

continue

else

for

long

signed

switch

void

case

default

enum

goto

register

sizeof

typedef

volatile

char

do

extern

if

return

static

union

while



Deklaracje zmiennych I

Język C zna następujące typy zmiennych:

1. **char** znakowe (elementarną jednostką informacji jest jeden znak).
2. **int** całkowite.
3. **float** zmiennoprzecinkowe („rzeczywiste”).
4. **double** zmiennoprzecinkowe, podwójnej precyzji.
5. **void** pusty typ; nie można zadeklarować zmiennej takiego typu, ale może być on wykorzystany do zwrócenia uwagi, że funkcja nie zwraca wartości lub, że nie przyjmuje parametrów.

Przykład deklaracji:

typ nazwa;



Deklaracje zmiennych II

```
int a;  
char b;  
float c;  
double d;
```



Wyprowadzanie wartości zmiennych

```
printf("Jakiś tekst %cośam tekst", zmienna);
```



char l

1. Zajmuje jeden bajt w pamięci komputera.
2. Typ służący do przechowywania znaków (ewentualnie liczb z zakresu 0...255 albo -128...127).
3. Każdy znak przechowywany jest jako odpowiadający mu kod ASCII.
4. Znaki zapisujemy w pojedynczych cudzysłowach (w odróżnieniu od tekstów — gdzie cudzysłowy są podwójne).
5. 'a' ; 'A' ; '7' ; '!' ; '\$'
6. Znaki „specjalne” (trudne do uzyskania z klawiatury!):
 - ▶ '\a' — alarm (sygnał akustyczny terminala),
 - ▶ '\b' — backspace (usuwa poprzedzający znak),
 - ▶ '\f' — wysunięcie strony (np. w drukarce),
 - ▶ '\r' — powrót kursora (karetki) do początku wiersza,
 - ▶ '\n' — znak nowego wiersza,



char II

- ▶ `'\"'` — cudzysłów,
- ▶ `'\''` — apostrof,
- ▶ `'\\'` — ukośnik wsteczny (backslash),
- ▶ `'\t'` — tabulacja pozioma,
- ▶ `'\v'` — tabulacja pionowa,
- ▶ `'\?'` — znak zapytania (pytajnik),
- ▶ `'\ooo'` — liczba zapisana w systemie oktalnym (ósemkowym), gdzie 'ooo' należy zastąpić trzycyfrową liczbą w tym systemie,
- ▶ `'\xhh'` — liczba zapisana w systemie heksadecymalnym (szesnastkowym), gdzie 'hh' należy zastąpić dwucyfrową liczbą w tym systemie,
- ▶ `'\unnnn'` — uniwersalna nazwa znaku, gdzie 'nnnn' należy zastąpić czterocyfrowym identyfikatorem znaku w systemie szesnastkowym. 'nnnn' odpowiada dłuższej formie w postaci 'ooooonnnn',
- ▶ `'\unnnnnnnn'` — uniwersalna nazwa znaku, gdzie 'nnnnnnnn' należy zastąpić ośmiocyfrowym identyfikatorem znaku w systemie szesnastkowym.



7. Napisy przechowujemy w tablicach typu znakowego! — Zmienne złożone.



Działania na zmiennych znakowych

- ▶ Takie jak na zmiennych całkowitych!
- ▶ Działania wykonywane są na wartościach ASCII odpowiadających literom.

Przykład

```
char a = 'B';  
char b = a / 2;  
char c = 'Z' / 2;  
char d = a + 1;
```



Drukowanie zmiennej znakowej

1. `putchar(zmienna_znakowa);`

2. albo

```
printf ("Wartość zmiennej znakowej wynosi = %c\n", zmienna_znakowa);
```



Przykład

```
#include <stdio.h>
int main ()
{
    char a = 'B';
    char b = a / 2;
    putchar(a); // albo printf('%c', a);
    putchar('\n');
    putchar(b); // albo printf('%c', b);
    putchar('\n'); // albo printf('\n', a);
    return 0;
}
```



1. Typ służy do przechowywania liczb całkowitych
2. Stałe całkowite:
 - ▶ Liczby można zapisywać w układzie dziesiętnym: 1; 123; 48; -579 itd.,
 - ▶ Liczby można zapisywać w układzie ósemkowym (zaczynając od zera): 01; 0123; 047; 0579 — w ostatnim przypadku będzie błąd!
 - ▶ Liczby można zapisywać w układzie szesnastkowym (zaczynając od 0x): 0xffff, 0xabcdef, 0Xabcd, 0xABCD.



Działania na liczbach całkowitych

1. Dodawanie (nic ciekawego): +,
2. Odejmowanie (nic ciekawego): −,
3. Mnożenie (nic ciekawego): * ,
4. Dzielenie (**uwaga na kłopoty**): / ,
 - ▶ wynik dzielenia dwu liczb typu całkowitego jest zawsze liczbą całkowitą!
 - ▶ $5 / 2$ w wyniku daje 2;
 - ▶ $2 / 3 * 3$ daje w wyniku 0 (zero!);
5. Reszta z dzielenia (modulo): ,
6. Działania na bitach:
 - ▶ iloczyn bitowy: & (&),
 - ▶ suma bitowa: | ,
 - ▶ różnica symetryczna: ^ ,
 - ▶ negacja: ~.



Dzielenie liczb całkowitych

I jeszcze raz: Są dwie operacje: $/$ i $\%$.

W przypadku argumentów całkowitych:

- ▶ Dzielenie wykonywane jest jako dzielenie całkowitoliczbowe!
- ▶ $3/5$ daje w wyniku 0
- ▶ $3/5 * 5$ daje w wyniku 0 (bo operacje wykonywane są od lewej do prawej z powodu jednakowych priorytetów mnożenia i dzielenia).
- ▶ $\%$ oznacza resztę z dzielenia całkowitoliczbowego

$$7 \% 3 = 1$$

bo

$$7 = 2 * 3 + 1$$

oraz

$$3 \% 7 = 3$$

bo

$$3 = 0 * 7 + 3$$



Jeszcze o dzieleniu liczb całkowitych

Jeszcze parę uwag o dzieleniu całkowitoliczbowym. Popatrzmy na taki program:

```
#include <stdio.h>
int main ( )
{
    for ( int i = b; i >= 0; i-- )
    {
        c = a / i;
        if ( c != poprzednie )
        {
            poprzednie = c;
            printf ( "%d/%d=%d\n", a, i, c );
        }
    }
    return 0;
}
```



Wyniki programu

50/100=0

50/50=1

50/25=2

50/16=3

50/12=4

50/10=5

50/8=6

50/7=7

50/6=8

50/5=10

50/4=12

50/3=16

50/2=25

50/1=50

Floating point exception (core dumped)



Operatory bitowe

1. $\&$ I (AND),
2. $|$ LUB (OR),
3. \wedge różnica symetryczna (exclusive OR),
4. \ll przesunąć w lewo,
5. \gg przesunąć w prawo,
6. \sim dopełnienie do jednego (przełącza wartość bitu: z 0 na 1, a z 1 na 0).

Operator jednoargumentowy!

Argumenty **całkowitoliczbowe** traktowane są jako ciągi bitów, a operacje wykonywane są na poszczególnych bitach.



Wyprowadzenie wartości całkowitej

1. Format dziesiętny: %d (%l — long, %h — short, %u — unsigned);
2. Format ósemkowy: %o;
3. Format szesnastkowy: %x;



float

1. Typ służy do przechowywania danych zmiennoprzecinkowych (zmiennopozycyjnych) czyli, tak zwanych rzeczywistych.
2. Liczby zapisujemy z **kropką** dziesiętną: 1.5f ; 23.387f ; 3.f
3. Albo używając zapisu „naukowego”: 3e6f ; 2.34E8f ; 3.14e-5F
4. Zwracam uwagę na literę f (F) na końcu! Bez niej stałe zmiennoprzecinkowe są traktowane jako typu **double**.
5. Zapis liczb zgodny z normą IEEE 754 na 32 bitach:
 - ▶ 7,22 cyfr dziesiętnych (23 bity),
 - ▶ zakres $10^{38,23}$.



Dwa programy

```
#include <stdio.h>
int main()
{
    float x = 10.3f;
    printf("%30.20g\n", x);
    return ( 0 );
}
```

10.300000190734863281



Dwa programy

```
#include <stdio.h>
int main()
{
    float x = 10.3f;
    printf("%30.20g\n", x);
    return ( 0 );
}
```

10.300000190734863281

```
#include <stdio.h>
int main()
{
    double x = 10.3;
    printf("%30.20g\n", x);
    return ( 0 );
}
```

10.3000000000000000711



Dwa programy

```
#include <stdio.h>
int main()
{
    float x = 10.3f;
    printf("%30.20g\n", x);
    return ( 0 );
}
```

10.300000190734863281

O liczbach typu **float** zapomnijmy!

```
#include <stdio.h>
int main()
{
    double x = 10.3;
    printf("%30.20g\n", x);
    return ( 0 );
}
```

10.3000000000000000711



double

1. Typ służy do przechowywania danych zmiennoprzecinkowych podwójnej precyzji. Liczba zajmuje podwójną ilość miejsca ale też jest dokładniejsza (więcej cyfr po przecinku).
2. Zapis liczb zgodny z normą IEEE 754 na 64 bitach:
 - ▶ 15,95 cyfr dziesiętnych (53 bity),
 - ▶ zakres $10^{307,95}$.
3. Oprócz typu **double** jest jeszcze **long double** (16 bajtów, 128 bitów). Stałe muszą być wyposażone w przyrostek L.



long double

```
#include <stdio.h>
int main()
{
    long double x = 10.3L;
    printf("%35.30Lg\n", x);
    return ( 0 );
}
```

10.3000000000000000000000000000001734723476



Operacje zmiennoprzecinkowe

1. dodawanie,
2. odejmowanie,
3. mnożenie,
4. dzielenie.



Wyprowadzanie liczb zmiennoprzecinkowych

1. format stałoprzecinkowy:
%f (albo %szerokosc_pola.liczba_cyfr_po_przecinkuf) na przykład %10.3f,
2. format zmiennoprzecinkowy: %e,
3. format „mieszany”: %g.



Wyprowadzanie liczb zmiennoprzecinkowych

```
#include <stdio.h>
int main()
{
    double y, x = 1.23456789;
    y = x;
    int i = 0;
    while ( i < 10 )
    {
        printf("%15g\t_\_%15g\n", x, y);
        x = x * 10;
        y = y / 10;
        i++;
    }
    return 0;
}
```



Wyprowadzanie liczb zmiennoprzecinkowych

1.23457	1.23457
12.3457	0.123457
123.457	0.0123457
1234.57	0.00123457
12345.7	0.000123457
123457	1.23457e-05
1.23457e+06	1.23457e-06
1.23457e+07	1.23457e-07
1.23457e+08	1.23457e-08
1.23457e+09	1.23457e-09



Modyfikatory (kwalifikatory) typów

- ▶ **short** — krótki
- ▶ **long** — długi

oraz

- ▶ **signed** — ze znakiem
- ▶ **unsigned** — bez znaku

Zatem **int**, **short int**, **long int**, **unsigned int**, **signed short int** i **unsigned long int** to zazwyczaj zupełnie różne typy danych (przechowują liczby z różnych zakresów).



Jak się w tym wszystkim połąpać?

1. Zapewne nie warto...
2. Ale trzeba wiedzieć, że taki problem jest!
3. Można napisać mały programik:

```
#include <stdio.h>

int main()
{
    printf("sizeof(char) = %d\n", sizeof(char));
    printf("sizeof(short) = %d\n", sizeof(short));
    printf("sizeof(int) = %d\n", sizeof(int));
    printf("sizeof(long) = %d\n", sizeof(long));
    printf("sizeof(float) = %d\n", sizeof(float));
    printf("sizeof(double) = %d\n", sizeof(double));
    printf("sizeof(long double) = %d\n", sizeof(long double));
    return 0;
}
```



Jak się w tym wszystkim połąpać?

W naszym laboratorium wyniki tego programu są następujące:

```
sizeof ( char   ) = 1
sizeof ( short  ) = 2
sizeof ( int    ) = 4
sizeof ( long   ) = 8
sizeof ( float  ) = 4
sizeof ( double ) = 8
sizeof ( long double ) = 16
```



Operatory arytmetyczne

- ▶ dodawanie: +
- ▶ odejmowanie: -
- ▶ mnożenie: *
- ▶ dzielenie: /
- ▶ reszta z dzielenie (modulo): % (tylko liczby całkowite!)
- ▶ zwiększenie ++
- ▶ zmniejszenie --



Operatory arytmetyczne (cd) I

Zwiększenie

1. Występuje w dwu wariantach:
 - ▶ przedrostkowym $++x$
 - ▶ przyrostkowym $x++$
2. Na pierwszy rzut oka nie ma różnicy. Zarówno $++x$ jak i $x++$ oznacza tyle co $x = x + 1$.
3. W programie może wystąpić tak:

```
int x;  
x++;  
++x;
```

4. W takim kontekście, po wykonaniu operacji



Operatory arytmetyczne (cd) II

Zwiększenie

```
int x, z;  
z = 6;  
x = 2;  
z = z/++x;
```

z przyjmie wartość 2, a x przyjmie wartość 3. Najpierw zwiększane jest x, a później wykonywane dzielenie.

5. A w takim kontekście, po wykonaniu operacji

```
int x, z;  
z = 6;  
x = 2;  
z = z/x++;
```



Operatory arytmetyczne (cd) III

Zwiększenie

z przyjmie wartość 3, a x przyjmie wartość 3. Najpierw wykonywane jest dzielenie, a później zwiększane x.



Operatory arytmetyczne (cd)

Zmniejszenie

1. Występuje w dwu wariantach:



Operatory arytmetyczne (cd)

Zmniejszenie

1. Występuje w dwu wariantach:
 - ▶ przedrostkowym – $-x$



Operatory arytmetyczne (cd)

Zmniejszenie

1. Występuje w dwu wariantach:
 - ▶ przedrostkowym $-x$
 - ▶ przyrostkowym $x-$



Operatory arytmetyczne (cd)

Zmniejszenie

1. Występuje w dwu wariantach:
 - ▶ przedrostkowym $-x$
 - ▶ przyrostkowym $x-1$
2. Na pierwszy rzut oka nie ma różnicy. Zarówno $-x$ jak i $x-1$ oznacza tyle co $x = x - 1$.



Operatory arytmetyczne (cd)

Zmniejszenie

1. Występuje w dwu wariantach:
 - ▶ przedrostkowym $-x$
 - ▶ przyrostkowym $x-$
2. Na pierwszy rzut oka nie ma różnicy. Zarówno $-x$ jak i $x-$ oznacza tyle co $x = x - 1$.
3. $-x-$ najpierw zmniejszane jest x i taka wartość bierze udział w kolejnej operacji (jeżeli występuje).



Operatory arytmetyczne (cd)

Zmniejszenie

1. Występuje w dwu wariantach:
 - ▶ przedrostkowym $-x$
 - ▶ przyrostkowym $x-$
2. Na pierwszy rzut oka nie ma różnicy. Zarówno $-x$ jak i $x-$ oznacza tyle co $x = x - 1$.
3. $-x-$ najpierw zmniejszane jest x i taka wartość bierze udział w kolejnej operacji (jeżeli występuje).
4. $x-$ aktualna wartość x bierze udział operacji (jeżeli taka występuje), a później zmniejszane jest x .



Operatory przypisania

1. Oprócz najzwyczajszego operatora przypisania (=) używanego w kontekście:

$$a = b$$

co czytamy zmiennej a przypisz wartość zmiennej b



Operatory przypisania

1. Oprócz najzwyczajszego operatora przypisania (=) używanego w kontekście:

$$a = b$$

co czytamy *zmiennej a przypisz wartość zmiennej b*

2. Występują operatory „złożone”

$$+ = \quad - = \quad * = \quad / = \quad \% = \quad \ll = \quad \gg = \quad \& = \quad ^ = \quad | =$$

stosowane w następujący sposób (\odot oznacza jeden z symboli $+$, $-$, $*$, $/$...)

$$a \odot = b$$

co czytamy się

$$a = a \odot b$$



Operatory logiczne

1. == równy



Operatory logiczne

1. `==` równy
2. `!=` nie równy



Operatory logiczne

1. `==` równy
2. `!=` nie równy
3. `>` większy



Operatory logiczne

1. `==` równy
2. `!=` nie równy
3. `>` większy
4. `<` mniejszy



Operatory logiczne

1. $==$ równy
2. $!=$ nie równy
3. $>$ większy
4. $<$ mniejszy
5. $>=$ większy lub równy



Operatory logiczne

1. $==$ równy
2. $!=$ nie równy
3. $>$ większy
4. $<$ mniejszy
5. $>=$ większy lub równy
6. $<=$ mniejszy lub równy



Operatory logiczne

1. == równy
2. != nie równy
3. > większy
4. < mniejszy
5. >= większy lub równy
6. <= mniejszy lub równy
7. && logiczne I (AND)



Operatory logiczne

1. == równy
2. != nie równy
3. > większy
4. < mniejszy
5. >= większy lub równy
6. <= mniejszy lub równy
7. && logiczne I (AND)
8. || logiczne LUB (OR)



Operatory logiczne

1. == równy
2. != nie równy
3. > większy
4. < mniejszy
5. >= większy lub równy
6. <= mniejszy lub równy
7. && logiczne I (AND)
8. || logiczne LUB (OR)
9. ! logiczne NIE (NOT)



Operatory logiczne

Uwagi

1. W języku C **nie ma** typu logicznego!



Operatory logiczne

Uwagi

1. W języku C **nie ma** typu logicznego!
2. Z definicji numeryczną wartością wyrażenia logicznego lub relacyjnego jest **1** jeżeli jest ono prawdziwe lub **0** jeżeli nie jest prawdziwe.



Operatory logiczne

Uwagi

1. W języku C **nie ma** typu logicznego!
2. Z definicji numeryczną wartością wyrażenia logicznego lub relacyjnego jest **1** jeżeli jest ono prawdziwe lub **0** jeżeli nie jest prawdziwe.
3. W operatorach logicznych **każda wartość różna od zera** traktowana jest jako prawda.



Operatory logiczne

Uwagi

1. W języku C **nie ma** typu logicznego!
2. Z definicji numeryczną wartością wyrażenia logicznego lub relacyjnego jest **1** jeżeli jest ono prawdziwe lub **0** jeżeli nie jest prawdziwe.
3. W operatorach logicznych **każda wartość różna od zera** traktowana jest jako prawda.
4. Operatory logiczne mają priorytet niższy od operatorów arytmetycznych; dzięki temu wyrażenie $i < lim-1$ jest rozumiane właściwie jako $i < (lim-1)$.



Operatory logiczne

&& i ||

1. Wyrażenia połączone tymi operatorami oblicza się od strony lewej do prawej.



Operatory logiczne

&& i ||

1. Wyrażenia połączone tymi operatorami oblicza się od strony lewej do prawej.
2. Koniec obliczania następuje natychmiast po określeniu wyniku jako „prawda” lub „fałsz”.



Operatory logiczne

&& i ||

1. Wyrażenia połączone tymi operatorami oblicza się od strony lewej do prawej.
2. Koniec obliczania następuje natychmiast po określeniu wyniku jako „prawda” lub „fałsz”.
3. Wiele programów korzysta z tego faktu. (**Hakerstwo!**).



Operatory logiczne

&& i ||

1. Wyrażenia połączone tymi operatorami oblicza się od strony lewej do prawej.
2. Koniec obliczania następuje natychmiast po określeniu wyniku jako „prawda” lub „fałsz”.
3. Wiele programów korzysta z tego faktu. (**Hakerstwo!**).
4. Priorytet operatora && jest wyższy od priorytetu operatora ||.



Operatory logiczne

&& i ||

1. Wyrażenia połączone tymi operatorami oblicza się od strony lewej do prawej.
2. Koniec obliczania następuje natychmiast po określeniu wyniku jako „prawda” lub „fałsz”.
3. Wiele programów korzysta z tego faktu. (**Hakerstwo!**).
4. Priorytet operatora && jest wyższy od priorytetu operatora ||.
5. Priorytety obu operatorów są niższe od priorytetów operatorów relacji i porównania.



Operatory logiczne

&& – przykład

```
i < lim -1 &&  
(c = getchar()) != '\n' &&  
c != EOF
```



Operatory logiczne

&& – przykład

```
i < lim -1 &&  
(c = getchar()) != '\n' &&  
c != EOF
```

1. Najpierw sprawdzamy czy `i < lim-1`.



Operatory logiczne

&& — przykład

```
i < lim -1 &&  
(c = getchar()) != '\n' &&  
c != EOF
```

1. Najpierw sprawdzamy czy $i < \text{lim}-1$.
2. Jeżeli nie — nie zostaną wykonane kolejne operacje.



Operatory logiczne

&& — przykład

```
i < lim -1 &&  
(c = getchar()) != '\n' &&  
c != EOF
```

1. Najpierw sprawdzamy czy $i < \text{lim}-1$.
2. Jeżeli nie — nie zostaną wykonane kolejne operacje.
3. Następnie czytany jest znak ($c = \text{getchar}()$)...



Operatory logiczne

&& — przykład

```
i < lim -1 &&  
(c = getchar()) != '\n' &&  
c != EOF
```

1. Najpierw sprawdzamy czy $i < \text{lim}-1$.
2. Jeżeli nie — nie zostaną wykonane kolejne operacje.
3. Następnie czytany jest znak ($c = \text{getchar}()$)...
4. ...a w kolejnym kroku sprawdza się czy nie jest to znak nowej linii $'\n'$
(Hakerstwo!)



Operatory logiczne

&& — przykład

```
i < lim -1 &&  
(c = getchar()) != '\n' &&  
c != EOF
```

1. Najpierw sprawdzamy czy $i < \text{lim}-1$.
2. Jeżeli nie — nie zostaną wykonane kolejne operacje.
3. Następnie czytany jest znak ($c = \text{getchar}()$)...
4. ...a w kolejnym kroku sprawdza się czy nie jest to znak nowej linii $'\n'$ (**Hakerstwo!**)
5. W ostatnim kroku sprawdza się czy wcześniej wczytany znak nie jest znakiem końca pliku (EOF)



Operatory inne

1. **sizeof** () wielkość obiektu/typu danych
2. **&** Adres (operator jednoargumentowy)
3. ***** Wskaźnik (operator jednoargumentowy)
4. **?** Wyrażenie warunkowe
5. **:** Wyrażenie warunkowe
6. **,** Operator serii



Kolejność operacji

1. () [] -> .
2. ! ~ ++ -- + - * & sizeof
3. * / %
4. + -
5. << >>
6. < <= >= >
7. == !=
8. &
9. ^
10. |
11. &&
12. ||
13. ?:
14. = += -= *= /= %= &= ^= |= <<= >>=
15. ,



Type casting I

Zasady

1. Automatycznie dokonywane są jedynie konwersje nie powodujące straty informacji (np. zamiana liczby całkowitej do float w wyrażeniu $f + i$).
2. Pewne operacje są niedozwolone (np. indeksowanie tablicy wskaźnikiem typu float) i żadna konwersja nie będzie wykonana.
3. Operacje powodujące utratę informacji (zmiana typu zmiennoprzecinkowego do całkowitego) mogą powodować ostrzeżenie, ale nie są zabronione.
4. Niejawne przekształcenia arytmetyczne działają na ogół zgodnie z oczekiwaniami. Gdy nie korzystamy z typu „unsigned” to w wyrażeniach arytmetycznych:
 - ▶ jeżeli jeden z argumentów jest long double, pozostały jest zamieniany do long double,
 - ▶ w przeciwnym przypadku jeżeli jeden z argumentów jest double drugi zostanie przekształcony do tego typu,



Type casting II

Zasady

- ▶ w przeciwnym razie jeżeli jeden jest typu float drugi zostanie przekształcony do tego typu.
 - ▶ w przeciwnym przypadku wszystkie argumenty char i short zostaną przekształcone do typu int
 - ▶ następnie jeżeli którykolwiek z argumentów ma kwalifikator long drugi zostanie przekształcony do tego typu.
5. Zwracam uwagę, że typ float nie jest automatycznie zmieniany do double
 6. W operacji podstawienia typ prawej strony jest przekształcany do typu lewej (co może powodować utratę informacji).
 7. Dłuższe (bitowo) liczby przekształcane są do krótszych przez odrzucenie bardziej znaczących bitów.
 8. W każdym wyrażeniu można wskazać sposób przekształcenia używając operatora cast (rzut): (nazwa-typu) wyrażenie.



Type casting

- ▶ `round()`: round to nearest integer, halfway away from zero
- ▶ `rint()`, `nearbyint()`: round according to current floating-point rounding direction
- ▶ `ceil()`: smallest integral value not less than argument (round up)
- ▶ `floor()`: largest integral value (in double representation) not greater than argument (round down)
- ▶ `trunc()`: round towards zero (same as typecasting to an int)

