Uvod u programiranje

- predavanja -

studeni 2025.

11. Operatori

Nije bilo vodiča…

Operatori: prioritet i asocijativnost

	OPERATOR	ASOCIJATIVNOST
↑ Viši prioritet	*poziv funkcije* `()` `[]` `.` `->` *sufiks* `++` `--`	L → D
	`!` `~` `sizeof` *adresa* `&` *indirekcija* `*` *prefiks* `++` `--` *unarni* `+` `-`	D → L
	`(cast)`	D → L
	aritmetički `*` `/` `%`	L → D
	binarni `+` `-`	L → D
	`<<` `>>`	L → D
	`<` `<=` `>` `>=`	L → D
	`==` `!=`	L → D
	bitovni `&`	L → D
	`^`	L → D
	`|`	L → D
	`&&`	L → D
	`||`	L → D
	`? :`	D → L
Niži prioritet ↓	`=` `*=` `/=` `%=` `+=` `-=` `&=` `^=` `|=` `<<=` `>>=`	D → L
	operator `,`	L → D

Isti simbol operatora za različite operacije

- U programskom jeziku C za neke od simbola operatora vrijedi da se isti simboli koriste za različite operacije. U takvim slučajevima vrsta operacije se određuje na temelju konteksta u kojem se simbol koristi
  - npr. simbol minus se koristi za dvije različite operacije

```
int a = 3, b = 5, c, d;
c = b - 5;          Oduzimanje, binarni operator
d = -b;             Negacija, suprotni predznak
```

  - iz tog razloga, simboli u tablici operatora, koji ovisno o kontekstu imaju različito značenje, dodatno su opisani, npr.
    - *unarni* `+ -`
    - *binarni* `+ -`
    - *operator* `,` (jer zarez predstavlja ili separator ili operator, ovisno o kontekstu)

Unarni +, -

- Unarni minus se koristi često
- Unarni plus je u jezik ugrađen uglavnom samo radi simetrije
  - ne obavlja ništa, osim implicitne konverzije operanada tipa char, short ili _Bool u tip int

```
float x = 5.f, y = -2.f;
     -x
     -x + -y
     -x - -y
     +x

int n = -5;
char c = 'A';
     +n
     +c
```

Rezultati izraza:
-5, float
-3, float
-7, float
5, float (operator + ovdje nema efekta)

Rezultati izraza:
-5, int (operator + ovdje nema efekta)
65, **int** (samo konverzija tipa, char→int)

Operacije na razini bitova (bitwise)

- Pristup do pojedinog bita ili grupe bitova
  - operandi moraju biti cjelobrojni!
  - uporaba: za "programiranje na niskoj razini", low-level programming
  - operacijski sustavi, driveri, mikrokontroleri, grafika na niskoj razini, kriptografija, programi u kojima su brzina i memorija kritični faktor
  - kompaktna pohrana podataka
    - npr. kako pohraniti 32 logičke vrijednosti uz najmanji mogući utrošak prostora?
      T F T F T T T T F F F T T F F T F F T F F F T F T T T F F F T T

```
_Bool podaci[32];
podaci[0] = 1;
podaci[1] = 0;
podaci[2] = 1;
...
podaci[31] = 1;
```
32 bajta

```
char podaci[32];
podaci[0] = 1;
podaci[1] = 0;
podaci[2] = 1;
...
podaci[31] = 1;
```
32 bajta

```
unsigned int podaci;
podaci = 0xAF1922E3;
```
4 bajta. Svaki bit registra pohranjuje jednu logičku vrijednost.

Problem: kako pristupiti do svake pojedine logičke vrijednosti (bita)?

Operacije na razini bitova (bitwise)

- Podsjetnik: logički operatori vrijednosti operanada koriste kao cjeline

```
int a = 13, b = 7;          Rezultati izraza:
    a && b                  1, int
    a || b                  1, int
    !a                      0, int
```

- Operatori na razini bitova djeluju na pojedinačne bitove operanada

```
int a = 13, b = 7;          AND na razini bitova
    a & b                   5, int
```

Objašnjenje:

$$
\begin{array}{rl}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_2 = 13_{10} \\
\& & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 = 7_{10} \\
 & \text{----------} \\
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101_2 = 5_{10}
\end{array}
$$

Operatori OR, XOR i NOT na razini bitova

```
int a = 5, b = 19;          OR na razini bitova
    a | b                   23, int
```

```
  0000 0000 0000 0000 0000 0000 0000 0101₂  =  5₁₀
| 0000 0000 0000 0000 0000 0000 0001 0011₂  = 19₁₀
  -----------------------------------------
= 0000 0000 0000 0000 0000 0000 0001 0111₂  = 23₁₀
```

```
int a = 21, b = 19;         XOR na razini bitova
    a ^ b                   6, int
```

```
  0000 0000 0000 0000 0000 0000 0001 0101₂  = 21₁₀
^ 0000 0000 0000 0000 0000 0000 0001 0011₂  = 19₁₀
  -----------------------------------------
= 0000 0000 0000 0000 0000 0000 0000 0110₂  =  6₁₀
```

```
int a = 21;                 NOT na razini bitova
    ~a                      -22, int
```

```
~ 0000 0000 0000 0000 0000 0000 0001 0101₂  =  21₁₀
= 1111 1111 1111 1111 1111 1111 1110 1010₂  = -22₁₀
```

Operatori posmaka bitova - right shift

- Operator posmaka bitova **u desno** (*right shift*) izračunava rezultat tako da binarni sadržaj lijevog operanda posmakne u desno za broj mjesta koji je određen desnim operandom
  - Bitovi na upražnjenim pozicijama na lijevoj strani popunjavaju se:
    - ako je lijevi operand tipa *unsigned int*: nulama
    - ako je lijevi operand tipa *signed int*: ovisno o vrijednosti prvog bita i implementaciji, nulama **ili** jedinicama
      - **stoga, radi prenosivosti (portabilnosti)**: za operaciju posmaka u desno za lijevi operand trebalo bi koristiti tip *unsigned int*

Operatori posmaka bitova - *right shift*

```
unsigned int a = 2147685213, b = 9;
    a >> b
```

posmak bitova u desno

4194697, unsigned int

```
    1000 0000 0000 0011 0001 0011 0101 1101₂        = 2147685213₁₀
>> 9 0000 0000 0100 0000 0000 0001 1000 1001₂        =    4194697₁₀
```

bitovi na upražnjenim pozicijama na lijevoj strani popunjavaju se nulama, bez obzira na vrijednost prvog bita lijevog operanda jer je lijevi operand tipa unsigned int

- numerička vrijednost rezultata operacije a >> b odgovara rezultatu cjelobrojnog dijeljenja $a : 2^b$

Operatori posmaka bitova - left shift

- Operator posmaka bitova **u lijevo** (*left shift*) izračunava rezultat tako da binarni sadržaj lijevog operanda posmakne u lijevo za broj mjesta koji je određen desnim operandom
  - Bitovi na upražnjenim pozicijama na desnoj strani uvijek se popunjavaju nulama

```
int a = 4957, b = 9;                    posmak bitova u lijevo
    a << b                              2537984, int
```

```
    0000 0000 0000 0000 0001 0011 0101 1101₂      =    4957₁₀
<< 9 0000 0000 0010 0110 1011 1010 0000 0000₂     = 2537984₁₀
```

bitovi na upražnjenim pozicijama na desnoj strani popunjavaju se nulama

  - numerička vrijednost rezultata operacije a << b odgovara rezultatu množenja $a \cdot 2^b$

VM3
SZ [2]1

Postavljanje pojedinačnih bitova

- Uz pretpostavku da se najmanje značajan bit nalazi na poziciji 0

Bit varijable **a** na poziciji **j** postaviti na 1 (bez promjene ostalih bitova)

```
a = a | 0x1 << j;
```

Primjer: bit na poziciji **5** varijable **a** postaviti na 1

```
a                0000 0000 0000 0000 0000 0001 1000 1001
0x1 << 5         0000 0000 0000 0000 0000 0000 0010 0000
a | 0x1 << 5     0000 0000 0000 0000 0000 0001 1010 1001
```

Bit varijable **a** na poziciji **j** postaviti na 0 (bez promjene ostalih bitova)

```
a = a & ~(0x1 << j);
```

Primjer: bit na poziciji **7** varijable **a** postaviti na 0

```
a                0000 0000 0000 0000 0000 0001 1000 1001
0x1 << 7         0000 0000 0000 0000 0000 0000 1000 0000
~(0x1 << 7)      1111 1111 1111 1111 1111 1111 0111 1111
a & ~(0x1 << 7)  0000 0000 0000 0000 0000 0001 0000 1001
```

Slide 12

VM3 Možda primjer za n-ti bit = v?
Vedran Mornar; 9.10.2018.

SZ [2]1 Počeo sam pisati primjer, ali mi je postalo glupo sve ponovo prepisivati kad sam shvatio da je to zapravo postavljanje bita na 0 iza kojeg slijedi postavljanje bita na v.
Slaven Zakošek; 6.11.2018.

"Čitanje" pojedinačnih bitova

- Uz pretpostavku da se najmanje značajan bit nalazi na poziciji 0 i da je varijabla a tipa unsigned int

Ispisati 0 ili 1, ovisno o bitu varijable **a** na poziciji **j**

```
printf("%d", a >> j & 0x1);
```

Primjer: ispisati bit varijable a na poziciji 3

```
a                0000 0000 0000 0000 0000 0001 1000 1001
a >> 3           0000 0000 0000 0000 0000 0000 0011 0001
0x1              0000 0000 0000 0000 0000 0000 0000 0001
a >> 3 & 0x1     0000 0000 0000 0000 0000 0000 0000 0001
```

Primjer: ispisati heksadekadsku vrijednost grupe 4 najmanje značajna bita varijable a

```
printf("%x", a & 0xF);
```

```
a                0000 0000 0000 0000 0000 0001 1000 1001
0xF              0000 0000 0000 0000 0000 0000 0000 1111
a & 0xF          0000 0000 0000 0000 0000 0000 0000 1001
```

Operatori uvećanja i umanjenja za 1

- Unarni operatori koji se koriste za skraćeno pisanje operacije uvećanja (++) ili umanjenja (`--`) za jedan
  - operand mora biti *modifiable lvalue*: varijabla skalarnog tipa, skalarni član polja ili strukture ...
- Ovi operatori izazivaju popratne efekte (*side effects*)
  - osim što po evaluaciji daju rezultat, također i mijenjaju sadržaj operanda. *Koji smo operator do sada upoznali koji također izaziva popratne efekte?*
- Postoji prefiksni i sufiksni oblik operatora. Uz pretpostavku da je `brojac` varijabla skalarnog tipa:

Izrazi s prefiksnim oblikom operatora

```
++brojac
--brojac
```

Izrazi s sufiksnim oblikom operatora

```
brojac++
brojac--
```

Prefiksni oblik

- vrijednost operanda se prvo uveća/umanji (dakle, prvo se dogodi popratni efekt), a tek zatim se evaluira rezultat izraza

```
int a = 5, b;
b = ++a * 10;
```

konačni rezultat: a = 6, b = 60

  - što se točno događa:
    - varijabla a se uvećava za 1 (popratni efekt operatora ++)
    - izračunava se rezultat izraza ++a (koristit će se u ostatku izraza) = 6
    - izračunava se 6 * 10 i pridružuje u varijablu b
    - konačni rezultat izraza pridruživanja je 60 (i taj se rezultat odbacuje)

```
b = ++a * 10;
```

jednaki efekt

⇔

```
a = a + 1;
b = a * 10;
```

Sve što je u vezi prefiksnog/sufiksnog oblika navedeno za operator ++ vrijedi i za operator --

Sufiksni oblik

- Rezultat izraza se prvo evaluira na temelju "stare" vrijednosti operanda, operand se uveća/umanji kasnije (nije specificirano točno kada, ali najkasnije prije dovršetka naredbe u kojoj se izraz koristi)

```
int a = 5, b;
b = a++ * 10;
```

konačni rezultat: a = 6, b = 50

  - što se točno dešava:
    - izračunava se rezultat izraza a++ (koristit će se u ostatku izraza) = 5
    - odmah sada ili kasnije, ali svakako prije dovršetka naredbe, uvećava se sadržaj varijable a
    - izračunava se 5 * 10 i pridružuje u varijablu b
    - konačni rezultat izraza pridruživanja je 50 (i taj se rezultat odbacuje)

```
b = a++ * 10;
```

jednaki efekt

⇔

```
b = a * 10;
a = a + 1;
```

Kada je prefiksni oblik == sufiksni oblik?

- Uočiti: ako se operator koristi samostalno (ne u složenijim izrazima kao u prethodnim primjerima), tada između djelovanja prefiksnog i sufiksnog oblika operatora nema razlike

```
++a;
```

```
a++;
```

jednaki efekt

⇔

```
a = a + 1;
```

- često se takav jednostavan izraz s operatorima uvećanja/umanjenja koristi u petljama s unaprijed poznatim brojem ponavljanja, za uvećanje ili umanjenje kontrolne varijable petlje

```
for (i = 1; i <= 10; ++i) {
   ...
```

ili `i++`, ovdje je svejedno

Izbjegavati nedefinirano ponašanje

- Višekratno (unutar iste naredbe) korištenje varijable na koju djeluje operator s popratnim efektom može dovesti do nedefiniranog rezultata, tj. moguće različitog rezultata za različite arhitekture i prevodioce

```
int i = 5, rez;
rez = i * ++i;
```

  - ovisno o tome hoće li se prvo izračunati lijevi ili desni operand u operaciji množenja (i ili ++i) rezultat će biti 30 ili 36. Ovisno o tome što se zapravo htjelo postići, ispravno bi bilo napisati jedno od:

```
++i;
rez = i * i;
```

```
rez = i * (i + 1);
++i;
```

Uvjetni (conditional) operator

- Uvjetni operator je (jedini) ternarni operator (koristi tri operanda)

Opći oblik

```
izraz1 ? izraz2 : izraz3
```

  - evaluira se (izračunava se) `izraz1`
  - ako je rezultat logička vrijednost istina, evaluira se `izraz2` i to je ukupni rezultat izraza (`izraz3` se u tom slučaju ne izračunava!)
  - inače, evaluira se `izraz3` i to je konačni rezultat izraza (`izraz2` se u tom slučaju ne izračunava!)
- Tipična primjena: pisanje kompaktnijeg programskog koda

```
if (x >= 0.0) {
   rez = x;
} else {
   rez = -x;
}
```

jednaki efekt

⇔

```
rez = x >= 0.0 ? x : -x;
```

Uvjetni (conditional) operator

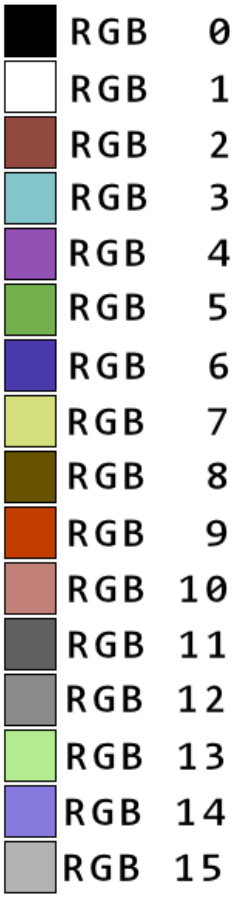
- Naročitu pažnju obratiti ako drugi ili treći izraz sadrže operatore s popratnim efektima

```
int i = 5, j = 10, rez;
rez = i < j ? ++i : ++j;
```
rez=6, i=6, **j=10** **izraz3 se nije evaluirao**

```
int i = 5, j = 10, rez;
rez = i > j ? ++i : ++j;
```
rez=11, **i=5**, j=11 **izraz2 se nije evaluirao**

Zadatak

- Koliko različitih boja imaju današnja računala?
- Commodore 64, 8b računalo je imalo svega 16 boja



- Kako bismo štedljivo poslali sliku s 4b bojama preko nekog „skupog” komunikacijskog kanala?
  - Jedna boja (pixel) je „pola bajta”
  - U int od 4B onda stane 8 boja!

Napišite program koji:

- učitava 8 boja (int),
- sprema ih u jedan int,
- raspakira ih natrag u 8 int varijabli

Skraćena evaluacija

- Na operatore s popratnim efektima također treba obratiti pažnju u složenim logičkim uvjetima zbog njihove skraćene evaluacije
  - skraćena evaluacija (*short circuit evaluation*) je svojstvo programskog jezika da dijelove logičkog izraza izračunava samo do trenutka kada se nepobitno utvrdi što će biti ukupni rezultat

```
int i = 5, j = 10, k = 15;
if (i > 5 && j == 10 && k < 15) ...
if (i > 4 || j == 10 || k < 15) ...
```

  - ako se npr. već prvi relacijski izraz evaluira kao laž (istina), preostali relacijski izrazi se ne evaluiraju
- Stoga, u sljedećem primjeru rezultat može biti neočekivan ako programer nije svjestan da C koristi skraćenu evaluaciju

```
int i = 5, j = 10, k = 15;
if (++i > 5 || ++j == 10 || ++k < 15) ...  ++j i ++k se neće obaviti
```

Operatori složenog pridruživanja

- operatori za složeno pridruživanje (*compound assignment*) koriste se za skraćeno pisanje izraza pridruživanja u kojem se nova vrijednost lijevog operanda (*modifiable lvalue*) izračunava na temelju njene stare vrijednosti primjenom binarnog aritmetičkog ili bitovnog operatora

```
duljina = duljina + a;
```
⇔
```
duljina += a;
```

- općenito, izraz pridruživanja u kojem se koristi jedan od binarnih operatora Ω (*, /, %, +, -, &, ^, |, <<, >>)

```
izraz1 = izraz1 Ω izraz2;
```

može se primjenom operatora složenog pridruživanja napisati ovako:

```
izraz1 Ω= izraz2;
```

Operatori složenog pridruživanja

- operatori su korisniji u slučajevima kada lijevi operand (*modifiable lvalue*) ima neki kompleksniji oblik (član polja, strukture i slično)

```
brojac[broj - D_GR] = brojac[broj - D_GR] + 1;
```

```
brojac[broj - D_GR] += 1;
```

- Oprez: u primjeni ovog operatora treba naročito paziti na prioritet operatora

```
a *= b + 5;
```

nije isto kao

```
a = a * b + 5;
```

- pri određivanju ekvivalentnog izraza najbolje je desnu stranu izraza složenog pridruživanja prepisati u zagradama

```
a *= b + 5;
```

jest isto kao

```
a = a * (b + 5);
```

Operator sizeof

Opći oblik

```
sizeof(izraz ili naziv tipa)
```

- Rezultat operacije je broj bajtova koji se koristi za pohranu operanda

```
float polje[10], x = 1.f;
double y = 2.;
char c = 'A';

        sizeof(double)                  8
        sizeof(x)                       4
        sizeof(x + y)                   8
        sizeof(polje)                  40
        sizeof('A' + 32)                4
        sizeof(unsigned short int)      2
        sizeof(0x100u)                  4
        sizeof(1LL)                     8
        sizeof(c)                       1
        sizeof(1.0L)                   12
```

Operator zarez

Opći oblik

```
izraz1, izraz2
```

- evaluira se `izraz1` (s rezultatom se dalje ne radi ništa)
- evaluira se `izraz2` i taj rezultat je ukupni rezultat izraza

- Ilustracija djelovanja operatora u jednom (besmislenom) primjeru

```
int i = 2, j = 5, k;
k = (++i, j = j * 2);
```

- evaluira se `++i`, vrijednost varijable `i` postaje `3`, rezultat se odbacuje
- izračunava se `j*2`, 10 se upisuje u j, rezultat tog pridruživanja (10) je ujedno ukupni rezultat djelovanja operatora *zarez*
- 10 se upisuje u varijablu `k`. Rezultat te operacije se odbacuje

Primjer

- Primjena više operatora *zarez* u istoj naredbi

```
int i, j, k, m;
m = (i = 5, j = i + 2, k = i + 3);
```

- zbog asocijativnosti operatora *zarez* (L→D) ekvivalentno je s

```
m = (((i = 5), j = i + 2), k = i + 3);
```

  - 5 se pridruži u `i`, rezultat pridruživanja 5 se odbaci
  - 7 se pridruži u `j`, rezultat pridruživanja 7 je ukupni rezultat prvog operatora *zarez*. Taj se rezultat odbacuje
  - 8 se pridruži u `k`, rezultat pridruživanja je 8, to je ukupni rezultat drugog operatora *zarez*. Taj se rezultat pridružuje varijabli `m`. Rezultat tog pridruživanja se odbacuje.

Operator zarez

- U praksi se ovaj operator koristi rijetko, obično u karakterističnim slučajevima
  - Primjer: ispisivati parove vrijednosti "uzlaznog i silaznog" brojača

```
int i, j;
j = 10;
for (i = 1; i <= 10; ++i) {
   printf("%d %d\n", i, j);
   --j;
}
```

Uz primjenu operatora *zarez*

```
int i, j;
for (i = 1, j = 10; i <= 10; ++i, --j) {
   printf("%d %d\n", i, j);
}
```

Primjer

- Još jedan primjer korištenja operatora *zarez*
  - učitavati cijeli broj i ispisivati njegov umnožak s 10 dok se ne upiše nula

```
int i;
do {
   scanf("%d", &i);
   if (i != 0) {
      printf("%d\n", 10 * i);
   }
} while (i != 0);
```

Uz primjenu operatora *zarez*

```
int i;
while (scanf("%d", &i), i != 0) {
   printf("%d\n", 10 * i);
}
```

Prije sljedećeg predavanja

- Edgar:
  - Tutorial: **Vodič „11 prije dvanaestog predavanja”**
  - **11. vježbe uz predavanja**

Zadatak

- Base64 kodiranje