Programarea Calculatoarelor
Cursul 3

Operatori pe biți
Instrucțiuni repetitive

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Obiective

- Care sunt operatorii care lucrează direct pe biți?
  - Când este recomandat să îi folosim?
- Cum scriem cod care repetă instrucțiuni?

Operatori pe biți

- Modifică direct biții din reprezentarea datelor
- Se folosesc doar cu tipuri întregi
- Cel mai util când biții unui număr reprezintă *n* variabile binare

- Complement față de 1 ~ (unar)
- Și logic pe biți & (binar)
- Sau logic pe biți | (binar)
- Sau exclusiv (xor) pe biți ^ (binar)
- Shiftare la stânga << (binar)
- Shiftare la dreapta >> (binar)

Operatori pe biți

- Complement față de 1 ~
  - schimbă starea fiecărui bit din reprezentarea numărului
  - pentru tipuri întregi cu semn: x + ~x = -1
    - fiindcă -x în complement față de 2 este ~x + 1
  - Utilitate: reprezentare complement față de 2
- Și logic pe biți &
  - Aplică operația de *și* bit cu bit pe cei doi operanzi
  - Pentru doi biți a & b este 1 doar dacă ambii biți sunt 1
  - Utilitate: verificare biți activi, schimbare biți specifici în 0
- Sau logic pe biți |
  - Aplică operația de *sau* bit cu bit pe cei doi operanzi
  - Pentru doi biți a | b este 1 dacă cel puțin unu dintre ei este 1
  - Utilitate: schimbare biți specifici în 1, reuniune stări

Operatori pe biți

- Sau exclusiv (xor) logic pe biți ^
  - Aplică operația de *sau exclusiv* (xor) bit cu bit pe cei doi operanzi
  - Pentru doi biți a ^ b este 1 dacă exact unu dintre ei este 1
  - Utilitate: schimbare starea unor biți specifici
- Shiftare la stânga x << k
  - Mută biții lui x cu k poziții la stânga și adaugă k biți de 0
  - De cele mai multe ori echivalent cu $x * 2^k$
  - Utilitate: înmulțire rapidă cu 2^k
- Shiftare la dreapta x >> k
  - Mută biții lui x cu k poziții la dreapta și păstrează semnul lui x
    - adaugă k biți de 0 pentru numere pozitive,
    - adaugă k biți de 1 pentru cele negative
  - De cele mai multe ori echivalent cu câtul împărțirii $x / 2^k$
  - Utilitate: împărțire rapidă cu 2^k

Operatori pe biți - identități utile

$1+2+4+...+2^{k-1} = 2^k-1$ este numărul cu primii k biți activați

$2^k = 1 << k$ este numărul care are doar bitul k activat

~0 = -1 0 nici un bit activat, -1 toți biții activați

0 | x = x 0 este element neutru pentru *sau*

~0 | x = ~0 ~0 sau transformă orice în ~0

0 & x = 0 0 și transformă orice în 0

~0 & x = x ~0 este element neutru pentru *și*

0 ^ x = x 0 este element neutru pentru *sau exclusiv* (xor)

x ^ x = 0 inversa unui element relativ la ^ este el însuși

Operatori pe biți - exemple

```
//exemple operatii
int a = 5;
int b = 9;
int c = 2;
printf("%d\n", a & b);
printf("%d\n", a ^ c);
printf("%d\n", a | b);
```

- considerăm reprezentările în binar ale variabilelor
- 5 = 0000 … $0101_{(2)}$
- 9 = 0000 … $1001_{(2)}$
- 2 = 0000 … $0010_{(2)}$
- 5 & 9 = 0000 … $0001_{(2)}$ = 1
- 5 ^ 2 = 0000 … $0111_{(2)}$ = 7
- 5 | 9 = 0000 … $1101_{(2)}$ = 13

```
//intregi cu si fara semn
printf("%d vs %u\n", ~0, ~0);
```

- numărul cu toți biții activați interpretat ca un întreg cu semn este -1, dar interpretat ca un întreg fără semn este cel mai mare număr pentru acel tip
- -1 vs 4294967295

```
//shiftare si overflow
printf("%d\n", 1 << 30);
printf("%d\n", 1 << 31);
printf("%u\n", 1 << 31);
```

- 1<<k = 2^k deci afișează 2^{30} = 1073741824
- depășește limita int-ului, overflow = -2^{31} = -2147483648
- ca întreg fără semn nu este overflow = 2^{31} = 2147483648

Operatori pe biți - exemple complexe

```
//shiftare numere negative
printf("%d\n", -1 >> 1);
```

- numărul cu toți biții activați shiftat la dreapta
- numerele negative se completează cu 1 la shiftare la dreapta pentru a păstra semnul
- -1 = $1111...1111_{(2)}$
- -1>>1 = $1111...1111_{(2)}$ = -1

```
printf("%d\n", -2 >> 1);
```

- -2 = $1111...1110_{(2)}$
- -2>>1 = $1111...1111_{(2)}$ = -1

```
printf("%d\n", -3 >> 1);
```

- -3 = $1111...1101_{(2)}$
- -3>>1 = $1111...1110_{(2)}$ = -2

```
int x;
scanf("%d",  &x);
if (x < 0){
        printf("%d\n", x >> 1);
        printf("%d\n", (x-1)/2);
}
```

- în general, x >> 1 pentru numere negative este (x-1)/2
- sau echivalent x/2 rotunjit în jos, parte întreagă

Instrucțiunea repetitivă while

- Execută în mod repetat instrucțiuni, condiționat de o expresie evaluată la începutul fiecărei iterații

```
while(expr)
        instr;
```

- Se evaluează expresia
  - Dacă este adevărată (diferită de 0) atunci se execută instrucțiunile din corpul buclei. După aceasta se trece din nou la pasul cu evaluarea expresiei.
  - Altfel nu se intră în buclă
- Echivalent cu "cât timp *expr* execută *instr*"
- Dacă avem mai multe instrucțiuni trebuie să le încadrăm între acolade {}

Instrucțiunea repetitivă while - exemplu simplu

```c
#include <stdio.h>

int main(){
        int n = 10;
        int s = 0;
        while(n > 0){
                s = s + n;
                n--;
        }
        printf("suma = %d\n",s);
        return 0;
}
```

- Calculează suma primelor *n* numere naturale, unde *n* >= *0*, *n* <= *1000*
- expresia din while este *n* > *0* = condiție de oprire
- instrucțiunile din while modifică valoarea expresiei care va deveni falsă la un moment dat
- Ce se întâmplă dacă *n* = *0*?
- Dacă *n* < *0*?
- Dacă *n* este un număr mare, de exemplu 10^9?

Instrucțiunea repetitivă do-while

- Execută în mod repetat instrucțiuni condiționat de o expresie evaluată la sfârșitul fiecărei iterații

```
do
        instr;
while(expr)
```

- Se execută instrucțiunile și apoi se evaluează expresia
  - Dacă este adevărată (diferită de 0) atunci se întoarce la pasul cu execuția instrucțiunilor
  - Altfel termină bucla
- Echivalent cu: "execută *instr*, dacă *expr* atunci repetă"
- Dacă avem mai multe instrucțiuni trebuie să le încadrăm între acolade {}

Instrucțiunea repetitivă do-while - exemplu simplu

```c
#include <stdio.h>

int main(){
        int n = 10;
        int s = 0;
        do{
                s = s + n;
                n--;
        }while(n>0);
        printf("suma = %d\n",s);
        return 0;
}
```

- Calculează suma primelor *n* numere naturale, unde *n* >= *0*, *n* <= *1000*
- instrucțiunile din do-while modifică valoarea expresiei care va deveni falsă la un moment dat
- expresia din do-while este *n* > *0* = condiție de oprire care se verifică după
- Ce se întâmplă dacă *n* = *0*?
- Dacă *n* < *0*?
- Dacă *n* este un număr mare, de exemplu 10^9?

Instrucțiunea repetitivă for

- Execută în mod repetat instrucțiuni, condiționat de o expresie evaluată la începutul fiecărei iterații

```
for(expr1; expr2; expr3)
        instr;
```

- Se evaluează *expr1*, o singură dată la început
  - este pentru inițializare
- Se verifică *expr2*, dacă este adevărată se intră în buclă altfel se trece la prima instrucțiune după for
- Se execută *instr*
- Se evaluează *expr3*
- Se trece din nou la pasul unde se evaluează *expr2*

Instrucțiunea repetitivă for

- Este echivalentul la "pentru valorile *expr1* până la *expr2* execută *instr* și apoi *expr3* în mod repetat"
- Chiar dacă lipsește o expresie trebuie incluse cele două ;
- Dacă avem mai multe instrucțiuni trebuie să le încadrăm între acolade {}
- Recomandat dacă avem nevoie de un contor pentru parcurgere
- Este echivalentă cu o buclă while

```
for(expr1; expr2; expr3)
        instr;
```

```
expr1;
while(expr2){
        instr;
        expr3;
}
```

Instrucțiunea repetitivă for - exemplu simplu

```c
#include <stdio.h>

int main(){
        int n = 10;
        int s = 0;
        for(int i=n; i>0; i--)
                s += i;
        printf("suma = %d\n",s);
        return 0;
}
```

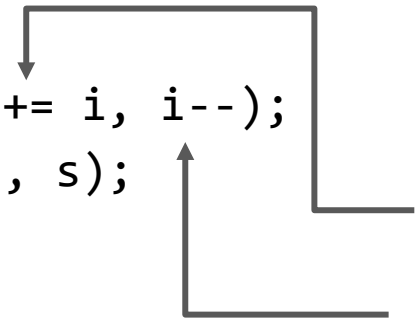
- Calculează suma primelor *n* numere naturale, unde *n >= 0, n <= 1000*
- expresia 1 declară și inițializează o variabilă locală i, valabilă doar în bucla for - recomandat
- expresia 2 este condiția de oprire este *i>0*
- conform expresiei 3, se decrementează *i* după fiecare iterație și înaintea verificării expresiei 2

- Ce se întâmplă dacă *n = 0*?
- Dacă *n < 0*?
- Dacă *n* este un număr mare, de exemplu 10^9?

Instrucțiunea repetitivă for - exemplu complex

```c
#include <stdio.h>

int main(){
        int n = 10;
        int s = 0;
        for(int i=n; i>0; s += i, i--);
        printf("suma = %d\n", s);
        return 0;
}
```

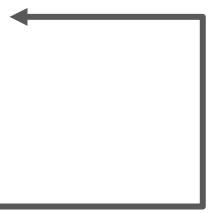


- Calculează suma primelor *n* numere naturale, unde *n >= 0, n <= 1000*
- am inclus adunarea la *s* în expresia 3 din for
- se folosește operatorul , pentru a separa instrucțiuni
- corpul buclei este gol, semnalat prin ; după for
- *s* trebuie declarat și initializat în afara for-ului

Instrucțiunea continue

- Sare peste instrucțiunile rămase din iterația curentă și continuă execuția buclei cu evaluarea expresiei
- Poate apărea doar în bucle
- Se referă la bucla cea mai apropiată

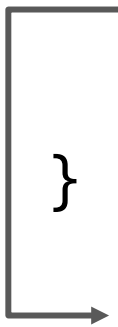
```
while(expr){
        instr1;
        continue;
        instr2;
}
```



Instrucțiunea break

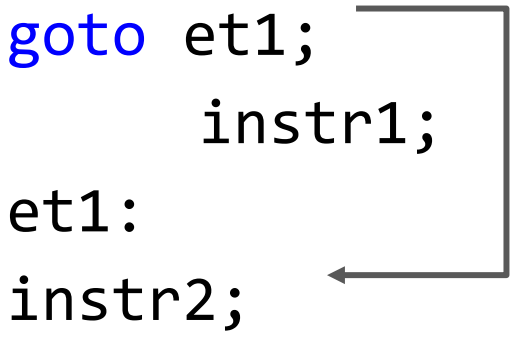
- Sare peste instrucțiunile rămase din iterația curentă și termină execuția buclei (întrerupe ciclul)
- Poate apărea doar în bucle (sau în instrucțiunea switch)
- Se referă la bucla cea mai apropiată

```
while(expr){
        instr1;
        break;
        instr2;
}
```



Instrucțiunea goto

- Sare la poziția indicată în cod de eticheta după goto
- O etichetă este un nume (identificator) urmat de :
- Utilă dacă se dorește ieșirea din mai multe bucle

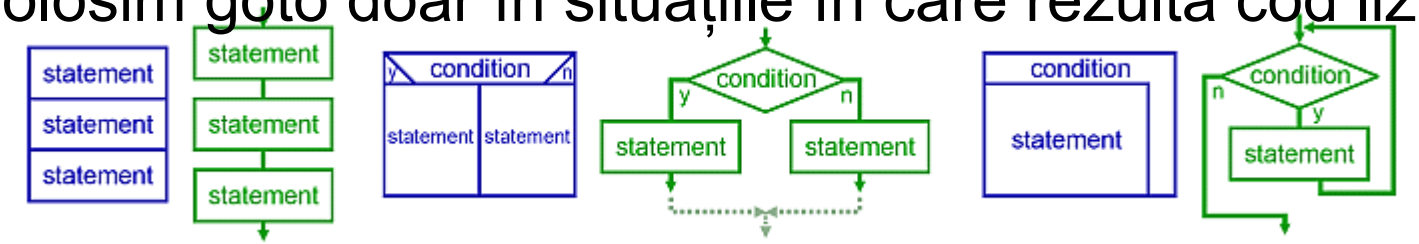


Funcția exit

- Întrerupe execuția programului și returnează codul de eroare trimis ca și parametru
  - funcționalitate echivalentă cu instrucțiunea `return` din funcția main dar poate fi apelată de oriunde
- Funcția se găsește în `stdlib.h`
- Este utilă dacă se întâmpină o eroare de la care nu se poate recupera

Programare structurată

- Este modul de programare care implică folosirea a doar 3 tipuri de instrucțiuni
  - secvențiale, de decizie și repetitive
- Teorema programării structurate - Un program care apelează subprograme în mod secvențial, sau condiționat de a valoare booleană sau în mod repetat poate calcula orice funcție computabilă
- Promovată de Edsger Dijkstra în 1968 pentru a evita folosirea instrucțiunilor de salt care produc cod dificil de urmărit
- Să folosim goto doar în situațiile în care rezultă cod lizibil



Analiză problemă - Cel mai mare divizor comun

- Se dau numerele naturale *a* și *b*, *0 <= a, b <= 10^7*
- Să se determine cel mai mare divizor comun al lui *a* și *b*
  - cel mai mare număr care îl divide și pe *a* și pe *b*
- Discutăm algoritmul care rezultă direct din definiție
  - există un algoritm mult mai eficient - algoritmul lui Euclid
- Notăm cu *d* divizorul căutat
- Observații
  - *1 <= d <= min(a, b)*
    - pentru că divizorul trebuie să fie mai mic sau egal ca *a* și mai mic sau egal ca *b*
      - nu se respectă dacă *a* sau *b* este 0
    - 1 este divizorul oricărui număr
  - *d* pentru *a = 0* și *b = 0* nu este clar definit
    - poate fi orice număr
    - stabilim să fie 0

Discuție problemă - Cel mai mare divizor comun

- Pseudocod

```
Citeste a, b
mn = min(a, b)
pentru d de la mn la 1
      daca a divizibil d si b divizibil d
            termina bucla
Afiseaza d
```

Discuție problemă - Cel mai mare divizor comun

```c
#include <stdio.h>

int main(){
        int a, b;
        scanf("%d%d", &a, &b);
        int mn = a < b ? a : b;
        int d;
        for(d = mn; d>0; d--){
                if (a%d == 0 && b%d == 0){
                        break;
                }
        }
        printf("cmmdc %d si %d este %d\n",
                a, b, d);
        return 0;
}
```

- calculăm minimul folosind operatorul ternar
- terminăm bucla dacă am găsit divizorul

- Rezultat incorect dacă mn = 0
- În acel caz trebuie d = max(a, b)

Discuție problemă - Cel mai mare divizor comun

```c
#include <stdio.h>

int main(){
        int a, b;
        scanf("%d%d", &a, &b);
        int mn = a < b ? a : b;
        int d = mn;
        while(d > 0 && (a % d || b % d)){
                d--;
        }
        if (mn == 0)
                d = a > b ? a : b;
        printf("cmmdc %d si %d este %d\n", a, b, d);
        return 0;
}
```

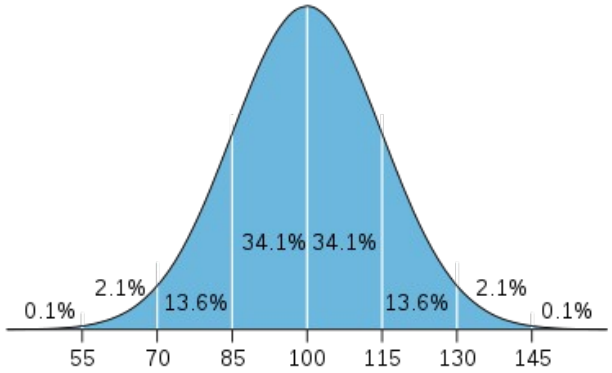
- *d* este inițializat, cu minimul
- cu buclă while
- cât timp *a* nu se divide la *d* sau *b* nu se divide la *d* descreștem *d*
- dacă minimul este 0 atunci cmmdc este maximul

Discuție problemă - Probabilitate IQ mare

- Care este probabilitatea ca dintre *n* persoane cel puțin una are IQ peste x

Discuție problemă - Probabilitate IQ mare

- Observații
  - vom presupune că distribuția de probabilitate este normală
  - folosind distribuția se poate calcula probabilitatea de a avea IQ <= x
    - pentru simplitate vom cere la intrare direct această probabilitate
  - este mai ușor să calculăm probabilitatea evenimentului complementar: nu există nici o persoană cu IQ mare



$$\frac{1}{15\sqrt{2\pi}} \int_{-\infty}^{130} e^{-\frac{(x-100)^2}{2 \cdot 15^2}} dx$$

citim n - numărul de persoane, p - probabilitate de IQ mic

calculăm q = p^n - probabilitate ca toți au IQ mic

afișăm 1-q

Discuție problemă - Probabilitate IQ mare

```c
#include <stdio.h>

int main(){
        int n;
        double p;
        puts("Nr. persoane?");
        scanf("%d", &n);
        puts("Probabilitate IQ <= x? [0, 1]");
        scanf("%lf", &p);
        double ans = 1;
        for(int i=0; i<n; i++)
                ans = ans * p;
        printf("Probabilitate ca cineva are
        IQ peste x: %.2f %%\n", (1-ans) * 100);
        return 0;
}
```

- modificăm cerința să avem *p* la intrare
- calculăm probabilitatea evenimentului complementar
- la afișare calculăm complementul și transformăm în procente