III. Conversii de tip

Conversii

- explicite
  - programatorul solicită ca o valoare de un anumit tip să fie convertită la alt tip
- implicite
  - expresii aritmetice care conţin variabile de tipuri diferite
  - apeluri de funcţii - tipul parametrului efectiv nu corespunde cu tipul parametrului formal

Tipuri slabe şi puternice (1)

a=b+c;

- variabilele b şi c sunt de tipuri diferite
- ce tip va avea rezultatul sumei?
  - unul din tipurile variabilelor b şi c
  - cel mai puternic dintre ele
  - dar care este acesta?
- ce înseamnă de fapt că un tip este mai puternic decât altul?

Tipuri slabe şi puternice (2)

- tipuri elementare
  - toate sunt numerice (inclusiv char)
  - fiecare poate reprezenta numere dintr-un anumit domeniu
  - tipul A este mai puternic decât B $\Leftrightarrow$ orice valoare reprezentabilă în B poate fi reprezentată şi în A
  - tipul A este mai puternic decât B $\Leftrightarrow$ domeniul tipului A include domeniul tipului B

Tipuri slabe şi puternice (3)

- tipul A este reprezentat pe mai mulţi octeţi decât tipul B
- înseamnă că A este mai puternic decât B?
  - sunt şanse bune să fie aşa
  - dar nu este obligatoriu
  - domeniul tipului A este mai mare decât domeniul tipului B
  - nu înseamnă automat că îl include complet

Exemple

- char

  -128 .. 127
- unsigned char

  0 .. 255
- int

  -2147483648 .. 2147483647
- unsigned int

  0 .. 4294967295

Exemple (cont.)

- tipul int este mai puternic decât char
- similar, unsigned int este mai puternic decât unsigned char
- şi int este mai puternic decât unsigned char
- dar unsigned int nu este mai puternic decât char

Cum facem deci conversiile?

- unde un tip este mai puternic decât celălalt - conversie la tipul mai puternic
- tipurile signed şi unsigned pe acelaşi număr de octeţi sunt considerate interschimbabile
  - între ele nu există practic conversie
  - doar copiere bit cu bit

Rezumat

unsigned char < unsigned short <
unsigned int (= unsigned long)

char < short < int (= long)

unsigned char < short

unsigned short < int

unsigned < float < double

int < float

Virgulă mobilă

- de fapt, tipul float nu este cu adevărat mai puternic decât int şi unsigned
  - există valori din aceste tipuri care nu pot fi reprezentate exact ca float
  - dar se consideră astfel, deoarece putem obţine o aproximare a valorii corecte
- tipul double este mai puternic decât toate celelalte

Cazuri speciale

- când nici un tip nu este mai puternic decât celălalt

```
unsigned u;
char c;
c=-20;
u=c;
printf("%u\n%d\n",u,u);
```

Cazuri speciale (cont.)

- rezultat afişat

```
4294967276
-20
```

- afişat ca întreg cu semn, u are aceeaşi valoare ca şi c
- s-a căutat un tip mai puternic decât char, echivalent cu unsigned int
  - deci char a fost convertit la int

Conversii explicite

- când avem nevoie de ele?
- atunci când modul implicit în care lucrează compilatorul cu tipurile nu produce efectul pe care îl dorim
- trebuie multă atenţie la ordinea în care se fac operaţiile

Exemplu 1

```
int i;
unsigned u;
if(i<u) ...
```

- le comparăm ca numere cu semn sau fără semn?
  - rezultatul comparaţiei poate diferi

```
if(i<(int)u) ...
```

Exemplu 2

```
int a,b;
float f;
a=5;
b=14;
f=(a+b)/2;
printf("%f\n",f);
```

- rezultat afişat: `9.000000`

Exemplu 2 (cont.)

- ordinea operaţiilor
  - întâi se adună `a` cu `b`
  - apoi se face împărţirea la 2
  - în final, rezultatul se depune în `f`
- la fiecare operaţie, operanzii trebuie să aibă acelaşi tip
  - altfel se realizează conversii implicite
- dar nu există o viziune globală asupra întregii expresii de calculat

Exemplu 2 (cont.)

- prima operaţie: `a+b`
  - ambii operanzi sunt de tip `int`
  - nu este necesară nici o conversie
  - rezultatul - de tip `int` (19)
- a doua operaţie: împărţirea la 2
  - ambii operanzi sunt de tip `int`
  - rezultatul - de tip `int` (9)
  - împărţirea se face prin trunchiere

Exemplu 2 (cont.)

- a treia operaţie: depunerea rezultatului în f
  - membrul stâng al atribuirii - de tip float
  - membrul drept - de tip int
  - membrul drept este convertit implicit la float
    - devine 9.0
  - trunchierea a fost făcută în pasul anterior
  - nu se mai poate reface valoarea corectă

Exemplu 2 (cont.)

- cum obţinem rezultatul corect?
- variantă incorectă

```
f=(float)((a+b)/2);
```

  - conversia la float se face tot prea târziu

- variantă corectă

```
f=((float)a+b)/2;
```

  - la fiecare operaţie, unul din operanzi va fi de tip float

Nivelul limbajului de asamblare

- conversiile cad în sarcina programatorului
- instrucţiunile procesorului necesită operanzi de aceeaşi dimensiune
  - aici nu contează tipul folosit în C/C++, ci dimensiunea sa

Conversie spre un tip mai slab (1)

```
int i;
char c;
//c=i;
_asm {
   mov al,byte ptr i
   mov c,al
}
```

Conversie spre un tip mai slab (2)

- similar pentru tipuri fără semn
- se poate pierde informaţie
  - exemplu: i = -2000 → c = 48
- dar nu avem o soluţie mai bună
  - valoarea -2000 nu poate fi reprezentată prin tipul char
  - dacă valoarea variabilei i intră în domeniul reprezentabil pentru char, variabila c o va prelua în mod corect

Conversie spre un tip mai puternic

- se poate face întotdeauna fără pierdere de informaţie
- tipuri fără semn - adăugarea de biţi 0 spre stânga
- tipuri cu semn - multiplicarea bitului de semn spre stânga

Tipuri fără semn - varianta 1

```
unsigned int i;
unsigned char c;
//i=c;
_asm {
   mov eax,0
   mov al,c
   mov i,eax
}
```

Tipuri fără semn - varianta 2

```
unsigned int i;
unsigned char c;
//i=c;
_asm {
   mov al,c
   movzx eax,al      //completare cu biti 0
   mov i,eax
}
```

Tipuri cu semn

```
int i;
char c;
//i=c;
_asm {
   mov al,c
   movsx eax,al    //extindere bit de semn
   mov i,eax
}
```

Pointeri (1)

- cum se face conversia între pointeri spre diferite tipuri?
- se poate face întotdeauna
  - pointerii au întotdeauna 4 octeţi

```
int i;
char *c;
c=(char*)&i;
```

Pointeri (2)

- în limbajul de asamblare
  - verificarea tipurilor se face doar când folosim numele variabilelor declarate în C/C++
  - în rest nu e necesară vreo conversie între pointeri

Exemplu

```
int i;
_asm {
   mov al,byte ptr i          //(1)
   inc al                     //(2)
   lea ebx,i                  //(3)
   mov [ebx],al               //(4)
}
```

Exemplu (cont.)

- linia 1
  - variabila i este accesată la nivel de octet
  - este necesară conversia (byte ptr)
    - compilatorul ştie că i este pe 4 octeţi
    - registrul al este pe 1 octet
    - deci operanzii ar avea dimensiuni diferite
  - este o conversie de pointer
    - un operand de la o adresă din memorie (i) va fi accesat cu altă dimensiune (deci alt tip)

Exemplu (cont.)

- liniile 3 şi 4
  - ebx devine pointer spre variabila i
  - când accesăm locaţia respectivă, nu mai folosim numele i
  - nu este necesară nici o conversie

IV. Referinţe şi pointeri

Ce este o referinţă

- nu apare în limbajul C
- introdusă în C++
- reprezintă un alt nume (alias) pentru o variabilă deja existentă

Declarare

```
int a;
int &b=a;
```

- b este o referinţă pentru a
- a nu se confunda cu operatorul adresă
- moduri incorecte de declarare:

```
int &b;
int &b=20;
```

Utilizare

```
int a;
int &b=a;
a=3;
b=5;
printf("%d\n",a);
```

- rezultat afişat: 5
- deci a fost modificată chiar variabila a

Reprezentare internă (1)

- ne putem gândi la două posibilităţi

a) compilatorul foloseşte două nume diferite pentru aceeaşi locaţie de memorie

b) b este de fapt un pointer către adresa variabilei a

Reprezentare internă (2)

```
a=3;
_asm {
        mov b,5
   }
printf("%d\n",a);
```

- rezultat afişat: 3
- dacă erau două nume pentru aceeaşi locaţie, s-ar fi afişat 5

Reprezentare internă (3)

```
a=3;
_asm {
        mov eax,b
        mov dword ptr [eax],5
   }
printf("%d\n",a);
```

- rezultat afişat: 5

Reprezentare internă (4)

- deci b este de fapt o variabilă separată
  - pointer - conţine adresa variabilei a
- compilatorul ascunde aceste detalii
- se doreşte să privim b ca pe un alt nume pentru a
- putem verifica prin afişarea adreselor variabilelor a şi b

Preluare adrese din C++

```
int a;
int &b=a;
int *p1,*p2;
p1=&a;
p2=&b;
printf("%p\n%p\n",p1,p2);
```

- rezultat afişare

```
0012FF60
0012FF60
```

- deci C++ raportează aceeaşi adresă

Preluare adrese din ASM

```
int a;
int &b=a;
int *p1,*p2;
_asm {
        lea eax,a
        mov p1,eax
        lea eax,b
        mov p2,eax
    }
printf("%p\n%p\n",p1,p2);
```

- rezultat afişare

```
0012FF60
0012FF54
```

- adresele sunt de fapt diferite

Utilitate (1)

- referinţele sunt deci pointeri
- avantaj - mai uşor de lucrat decât cu pointerii
  - nu mai trebuie să folosim operatorii & (adresă) şi * (dereferenţiere) → mai puţine greşeli
- dezevantaj - mai puţin flexibile
  - o referinţă este legată de o singură variabilă pe toată durata existenţei sale

Utilitate (2)

- referinţele nu pot deci înlocui pointerii în toate situaţiile
  - exemplu - parcurgerea elementelor unui tablou

```
char s[]="abcd",*p;
for(p=s;*p!='\0';p++) // ...
```

- unde sunt totuşi utile?
- cel mai des folosite - parametri pentru funcţii

Referinţe ca parametri (1)

```
void f(int &m) {//...}
int x;
f(x);
```

- parametrul m este o referinţă pentru variabila x
- orice modificare asupra lui m se reflectă asupra lui x → transfer prin referinţă

Referinţe ca parametri (2)

- atenţie la ce înseamnă referinţă

```
void f(int &m) {//...}
f(5);
```

- eroare la compilare - referinţa este o adresă
- la fel ca în cazul pointerilor

Referinţe vs. pointeri (1)

- care este de preferat?
- unde le putem folosi pe ambele - referinţe
  - sintaxă mai simplă
- în unele situaţii putem folosi doar pointeri
- cum stau lucrurile la nivelul limbajului de asamblare?
  - nici o diferenţă între referinţe şi pointeri

Referinţe vs. pointeri (2)

```
void add_3(int *x)
{
_asm {
  mov eax,x
  add dword ptr [eax],3
  }
}
```

```
void add_3(int &x)
{
_asm {
  mov eax,x
  add dword ptr [eax],3
  }
}
```

Pointeri dubli

- un pointer conţine adresa unei variabile
- această variabilă poate fi la rândul său un pointer
  - ş.a.m.d. (foarte rar)
- trebuie multă atenţie la tratarea lor

Exemplu

- o funcţie pentru inserarea unui element într-o listă înlănţuită
- parametru - capul listei
- ce tip are?
  - pointer
  - dar dacă elementul trebuie inserat înainte de primul din listă?
  - modificăm capul listei - ne trebuie adresa sa

Mai des decât credem

- în limbajul de asamblare, pointerii dubli apar frecvent

```
void f(int *x)
mov eax,[ebp+8] // [ebp+8] - pointer dublu
mov dword ptr [eax],5 // *x=5;
```

- eroare comună

```
mov dword ptr [ebp+8],5
```