Adattípusok C-ben

Kormányos Andor

Komplex Rendszerek Fizikája Tanszék

2019. szeptember 16.

Motiváció

- valós számok ábrázolása a számítógépen mindig közelítést igényel
- ez fontos lehet pl ha nagyon kicsi vagy nagyon nagy számokkal végzünk műveleteket: olyan hibák léphetnek fel, amelyek az matematikai számolásokban ismeretlenek
- minél több bitet használunk számábrázolásra, annál pontosabb lehet a számítás de annál nagyobb a memóriaigény

Adattípusok a számítógépben

Legkisebb információhordozó egység: bit ("binary digit"): 0 vagy 1 érték (pl. a computer memóriájában egy cella tartalmaz töltést v nem, a hard disk egy cellájában a mágnesezettség stb)

A memória alapegysége a bájt (byte)

- 1 bájt = 8 bit 0 vagy 1 értékkel: $2^8 = 256$ különböző érték
- ez kb. **betűk** és **karakterek** tárolására elegendő
- minden bájtnak közvetlen **memóriacíme** van

Hogy tárolhatók 256-nál nagyobb egész számok?

- több bájtot kell egybefogni
- 2 bájt = 16 bit = 65536 különböző érték
- 4 bájt = 32 bit = kb. 4 md különböző érték
- ha szükséges, 1 bit használható előjelnek
- ilyenkor az ábrázolható tartomány kb. $\pm 2^{b-1}$ lesz

Adattípusok a számítógépben

Hogyan tárolhatók törtszámok?

- **lebegőpontos** alakban (floating point)
- a lebegőpontos műveleteket a processzor hardveresen valósítja meg

Számábrázolás

Tízes számrendszer:

- $107 = 10^2 \cdot 1 + 0 \cdot 10^1 + 7 \cdot 10^0$
- minden 10-hatvány együtthatója tíz értéket vehet fel

Számábrázolás

Tízes számrendszer:

- $107 = 10^2 \cdot 1 + 0 \cdot 10^1 + 7 \cdot 10^0$
- minden 10-hatvány együtthatója tíz értéket vehet fel

Számítógép: bináris számrendszer (sokkal hatékonyabb hardveres szintű műveletek)

- számábrázolás alapja: kettes számrendszer
- a 2 hatványait használjuk

$107 = 64 + 32 + 8 + 2 + 1$

0	1	1	0	1	0	1	1
2^7	$2^6 (= 64)$	$2^5 (= 32)$	2^4	$2^3 (= 8)$	2^2	$2^1 (= 2)$	$2^0 (= 1)$

- pozitív egész számok ábrázolására
- 1 bájt: $0 - 255$
- 2 bájt: $0 - 65535$, stb.
- egy bitet el lehet használni előjelnek, ekkor az ábrázolható legnagyobb szám megfeleződik

Fixpontos ábrázolás

Hasonlóan működhetne a törtszámok ábrázolása

$1.625 = 1 + 1/2 + 0 + 1/8$

0	0	0	1	1	0	1	0
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

- az ábrázolható tartomány nagyon kicsi
- van egy legkisebb lépésköz (felbontás)

Lebegőpontos ábrázolás

10-es számrendszerben gyakran használjuk a normálalakot

- véges sok tizedest adunk meg, valamint egy kitevőt:

$$+2{,}3454612 \times 10^3 \qquad (+0.23454612\text{E}+4)$$

- előjel
- mantissza
- kitevő

Lebegőpontos számok kettes számrendszerben

$$(-1)^s \cdot \underbrace{(1 + \sum_{i=1}^{t} x_i 2^{-i})}_{f} \cdot 2^{e-B}$$

- s: előjel (1 bit)
- f: mantissza
- e: kitevő v. karakterisztika
- B: konstans

Lebegőpontos számok kettes számrendszerben

$$(-1)^s \cdot \underbrace{(1 + \sum_{i=1}^{t} x_i 2^{-i})}_{f} \cdot 2^{e-B}$$

- s: előjel (1 bit)
- f: mantissza
- e: kitevő v. karakterisztika
- B: konstans

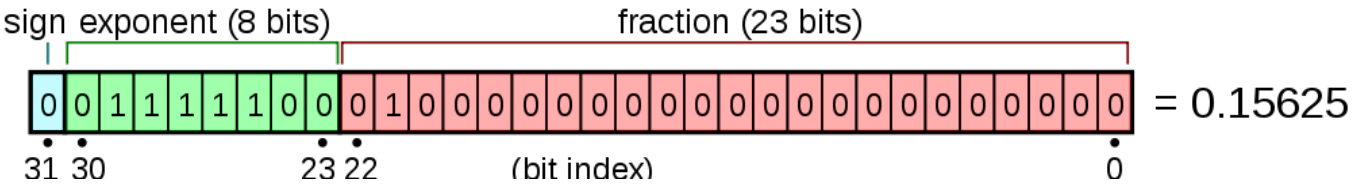
Feladat

Írjuk fel a 0.1-t 2-es számrendszerbeli lebegőpontos számként a fenti alakban! Mit tapasztalunk?

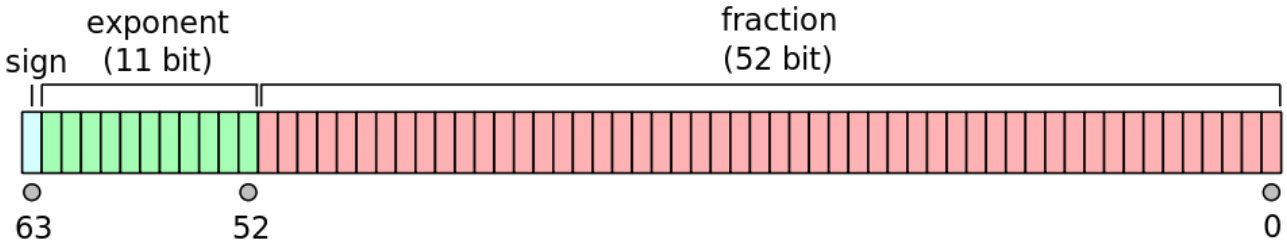
IEEE 754 szabvány

32, illetve 64 bitet kell kiosztani s, f és e tárolására

- IEEE 754 szabvány
- float: 32 bit, 0 körül kb. 7 tizedes pontosság



- double: 64 bit, 0 körül kb. 16 tizedes pontosság



- speciális értékek: $\pm\infty$, NaN, előjeles nulla

Lebegőpontos ábrázolás felbontása

A problémát illusztrálhatjuk a következő egyszerű példával.

Az egyszerűség kedvéért tegyük fel, hogy 10-es számrendszerben dolgozunk. Egy lebegőpontos számot akarunk ábrázolni, három helyjegy használható a mantisszára és egy jegy az exponensre:

$$mantissa \times 10^{exponens}$$

Lebegőpontos ábrázolás felbontása

A problémát illusztrálhatjuk a következő egyszerű példával.

Az egyszerűség kedvéért tegyük fel, hogy 10-es számrendszerben dolgozunk. Egy lebegőpontos számot akarunk ábrázolni, három helyjegy használható a mantisszára és egy jegy az exponensre:

$$mantissa \times 10^{exponens}$$

Ha az exponens 0, akkor a minden egész szám a $0 - 999$ intervallumban pontosan ábrázolható.
Ha az exponens 1, akkor az előbbi számhalmaz minden tagját lényegében megszorozzuk 10-el, tehát a $0 - 9990$ intervallumba eső számokat kapunk. Most azonban csak a 10 többszörösei reprezentálhatóak pontosan, mert továbbra is csak 3 helyjegy használható a mantisszára.

Lebegőpontos ábrázolás felbontása

A problémát illusztrálhatjuk a következő egyszerű példával.

Az egyszerűség kedvéért tegyük fel, hogy 10-es számrendszerben dolgozunk. Egy lebegőpontos számot akarunk ábrázolni, három helyjegy használható a mantisszára és egy jegy az exponensre:

$$mantissa \times 10^{exponens}$$

Ha az exponens 0, akkor a minden egész szám a $0 - 999$ intervallumban pontosan ábrázolható.
Ha az exponens 1, akkor az előbbi számhalmaz minden tagját lényegében megszorozzuk 10-el, tehát a $0 - 9990$ intervallumba eső számokat kapunk. Most azonban csak a 10 többszörösei reprezentálhatóak pontosan, mert továbbra is csak 3 helyjegy használható a mantisszára.
Hasonló probléma lép fel akkor is, ha 2-es számrendszer dolgozunk, a nulla körül a legpontosabb a számábrázolas.

Lebegőpontos felbontás hatása a műveletek eredményére

Például:

- ha a két szám között sok nagyságrend eltérés van, sokat romolhat a pontosság
- egyes esetekben azonos nagyságrendű számok kivonásánál is romolhat a pontosság

Lebegőpontos felbontás hatása a műveletek eredményére

Például:

- ha a két szám között sok nagyságrend eltérés van, sokat romolhat a pontosság
- egyes esetekben azonos nagyságrendű számok kivonásánál is romolhat a pontosság

További olvasnivaló (sok részlet, példák):
David Goldberg:
What every computer scientist should know about floating-point arithmetic

Számtípusok a C nyelvben

Egész számok

- `char` - 1 bájt
  előjel nélküli egész
- `short` - 2 bájt
  előjeles egész
- `int` - 4 bájt
  előjeles egész
- `long` - 4 bájt
  előjeles egész
- `long long` - 8 bájt
  előjeles egész

Előjeles/előjel nélküli egészek, pl.:

- `signed char`: $[-127 : 127]$
- `short`: $[-32767 : 32767]$
- `unsigned short`: $[0 : 65535]$

Lebegőpontos számok (floating point)

- `float` - 4 bájt
  kb. 7 jegy pontos
- `double` - 8 bájt
  kb. 16 jegy pontos
- `long double` - 10 bájt
  kb. 19 jegy pontos (ritkán használt)

Műveletek számokkal: összeadás, kivonás

Az eredmény típusa függ az operandusok típusától

- ha az operandusok azonos típusúak, az eredmény is olyan típusú lesz
- pl.: `int` + `int` = `int`
- pl.: `float` + `float` = `float`

- kivéve, ha valamelyik kevésbé pontos, mint `int`, mert akkor a fordító előbb `int`-re konvertál
- pl.: `short` + `char` = `signed int`

- kivéve, ha valamelyik pontosabb, mint a másik, mert akkor a fordító előbb a pontosabbra konvertál
- pl.: `int` + `double` = `double`
- pl.: `float` + `double` = `double`

Műveletek számokkal: összeadás, kivonás, szorzás

Mi történik, ha kifutunk az ábrázolt tartományból?

- pl. két `short` számot összeadva 32756-nél nagyobb eredmény adódik?
- ilyenkor **csendes túlcsordulás** történik

```
1  int main() {
2    short a = 55000, b = 25000;
3    short s = a + b;
4    printf("%d\n", s);   // eredmeny: 14464
5    return 0;
6  }
```

Ilyenkor nem kapunk semmilyen "hibaüzenetet"

- fordításkor sincsen figyelmeztetés
- tudni kell róla, hogy ilyen történhet, és ha ez gond, akkor megfelelően ellenőrizni kell a számokat

Műveletek egész számokkal: osztás

Az osztás kivezet az egész számok értékkészletéből

- ilyenkor nem történik automatikus típuskonverzió
- helyette: **egészosztás művelet lefelé kerekítéssel**

```
1  int main() {
2    int a = 12, b = 5;
3    printf("%d\n", a / b);   // = 2
4    return 0;
5  }
```

Ha **lebegőpontos osztást** szeretnénk

- legalább az egyik operandus legyen `float` vagy `double`
- ehhez lehet, hogy **konvertálni** kell.

```
1  int main() {
2    int a = 12, b = 5;
3    printf("%f\n", (double)a / b);   // = 2.400000
4    return 0;
5  }
```

Típusok lefelé konvertálása

Ha különböző típusú operandusokon végzünk műveletet

- a fordító a pontosabb típus irányába konvertál
- ez a konverzió **implicit**, nem kell semmit sem kiírni

Mi a helyzet akkor, ha eredményül mi kevésbé pontos számot szeretnénk?
Pl egy `double` eredmény konvertálása `float`-ra, hogy a lemezen kevesebb helyet foglaljon

Típusok lefelé konvertálása

Ha különböző típusú operandusokon végzünk műveletet

- a fordító a pontosabb típus irányába konvertál
- ez a konverzió **implicit**, nem kell semmit sem kiírni

Mi a helyzet akkor, ha eredményül mi kevésbé pontos számot szeretnénk? Pl egy `double` eredmény konvertálása `float`-ra, hogy a lemezen kevesebb helyet foglaljon

- ilyenkor **explicit** konvertálás, azaz **cast**-olás szükséges
- pl. számolás után kerekítés, majd integerre konvertálás

```
1  int main() {
2    // osztas szabalyos kerekitessel
3    printf("%d\n", (int)(17.24 / 5.1 + 0.5));
4    // eredmeny: 3
5    return 0;
6  }
```

Típusok lefelé konvertálása

Ha különböző típusú operandusokon végzünk műveletet

- a fordító a pontosabb típus irányába konvertál
- ez a konverzió **implicit**, nem kell semmit sem kiírni

Mi a helyzet akkor, ha eredményül mi kevésbé pontos számot szeretnénk? Pl egy `double` eredmény konvertálása `float`-ra, hogy a lemezen kevesebb helyet foglaljon

- ilyenkor **explicit** konvertálás, azaz **cast**-olás szükséges
- pl. számolás után kerekítés, majd integerre konvertálás

```
1  int main() {
2    // osztas szabalyos kerekitessel
3    printf("%d\n", (int)(17.24 / 5.1 + 0.5));
4    // eredmeny: 3
5    return 0;
6  }
```

Figyelem!

- a `(int)` művelet csak a közvetlenül utána álló operandusra vonatkozik
- ha az egész kifejezést castolni kell, akkor zárójel szükséges

Számok összehasonlítása

A processzor két változó értékét bitenként hasonlítja össze

- azonos integer típusok esetében tehát nincsen probléma
- különböző integer típusok esetén implicit konverzió történik

A lebegőpontos számok esetében már lehetnek gondok

- a műveletek nem végtelenül pontosak: kerekítési hiba
- a konstansokat 10-es számrendszerben adjuk meg, de a gépben minden bináris

```
1  int main() {
2    double a = 0.1 + 0.2;
3    if (a == 0.3) {
4      printf("egyenlo\n");
5    } else {
6      printf("nem egyenlo\n");
7      printf("    %.17f\n", 0.3);
8      printf("a = %.17f\n", a);
9    }
10   return 0;
11 }
```

Az előző program kimenete

```c
int main() {
  double a = 0.1 + 0.2;
  if (a == 0.3) {
    printf("egyenlo\n");
  } else {
    printf("nem egyenlo\n");
    printf("    %.17f\n", 0.3);
    printf("a = %.17f\n", a);
  }
  return 0;
}
```

Kimenet:

```
nem egyenlo
    0.29999999999999999
a = 0.30000000000000004
```

Feladat

A fenti jelenség jobb megértéséhez írjuk fel a 0.3-t is 2-es számrendszerbeli lebegőpontos alakba!

Megoldás lebegőpontos számok összehasonlítására

Be kell vezetnünk egy ϵ precizitást

- minden összehasonlításnál ekkora eltérést engedünk

```c
int main() {
  double epsilon = 1e-7;
  double a = 0.1 + 0.2;
  if (fabs(a - 0.3) < epsilon) {
    printf("majdnem egyenlo\n");
  } else {
    printf("nem egyenlo\n");
    printf("    %.17f\n", 0.3);
    printf("a = %.17f\n", a);
  }
  return 0;
}
```

Kimenet:

```
majdnem egyenlo
```

Osztás nullával vagy nagyon kicsi számmal

Problémák:

- nullával nem szabad osztani
- nagyon kicsi számmal igen, de lehet, hogy az eredményt már nem képes ábrázolni a `double`

Megoldás:

- a `double` szabvány előír néhány speciális értéket
- `NaN`: not-a-number, nem szám
- `Inf`: infinity, végtelen
- előjeles nulla, stb.

```
1  int main() {
2    printf("%.17f\n", 0.0 / 0.0);
3    printf("%.17f\n", 1.0 / 0.0);
4    return 0;
5  }
```

Kimenet:

```
1  -1.#IND0000000000000
2  1.#INF0000000000000
```

Logikai kifejezések

Az összehasonlító operátorokkal már találkoztunk

- <, >, ==, !=
- ezek eredménye 0 vagy 1, azaz egy integer szám!

Logikai operátorok: logikai kifejezések között értelmezett operátorok

- és (&&), vagy (||), nem (!) és a zárójelek
- (ezekből bitenként értelmezett változat is van, de azok máshogy működnek)

A logikai operátorokkal összekapcsolt kifejezések kiértékelése

- a || $\prec$ && $\prec$! precedencia szerint
- de **a kiértékelés csak addig tart, amíg az eredmény nem egyértelmű**!

```
1  int x, y, z;
2  if ( x < y && y < z )
3    printf("x is less than z\n");
```

A fenti példában

- ha $x < y$ teljesül, még $y < z$-t is le kell ellenőrizni
- ha $x < y$ nem teljesül, akkor $y < z$-t már mindegy

Még pár operátor

Inkrementáló operátorok

- `i++` – post-increment
  a változó értékét eggyel növeli, de csak miután a kifejezés kiértékelődött
- `-i` – pre-decrement
  a változó értékét eggyel csökkenti, de csak miután a kifejezés kiértékelődött

Értékadó operátorok

- `a += 5` – a változó értékét 5-tel növeli
- `b *= 2` – hasonló

Számok kiírása szövegként - printf

A printf függvény működése:

- az első paramétere egy formátum-string
- a formátumban %-kal kezdődő *tokenek* vannak
- lehetnek még benne \-lel kezdődő speciális karakterek
- a formátumot követően tetszőleges számú bemenő paraméter állhat

```
1  int main() {
2    int i = 2;
3    double d = 13.274324234;
4    float f = 14.545453f;
5    printf("%d -- %.3f -- %f3.2\n", i, d, f);
6  }
```

A program kimenete

```
1  2 -- 13.274 --  14.55
```

A formátumstringek felépítése (kivonat)

A tokenek `%n.mb` alakúak, ahol

- `n` egy szám, ami nem kötelező
  - ha pozitív, akkor a szám jobbra lesz igazítva, összesen `n` karaktert kihasználva
  - ha negatív, akkor a szám balra lesz igazítva, a végén szóközökkel kiegészítve
- `.m`, ahol `m` egy szám, ez sem kötelező
  - csak pozitív lehet
  - megadja, hogy hány tizedes kerüljön kiíratásra
- `b` egy betű, ami a paraméter típusától függ
  - `d` – integer
  - `u` – előjel nélküli integer
  - `f` – lebegőpontos szám tizedestört alakban
  - `e` – lebegőpontos szám exponenciális alakban

A formátumstringek felépítése (kivonat)

Fontosabb speciális karakterek (escape)

- `\n` – újsor-karakter (linux)
- `\r` – vissza a sor elejére karakter
- `\r\n` – újsor-karakter (windows)
- `\t` – tabulátor
- `\"` – idézőjel (különben a formátumstring végét jelentené)
- `\\` – egy darab `\`