E-02: ESPRESSIONI E NUMERI IN C

Fondamenti di Informatica e Laboratorio T-AB

Corso di Laurea in Ingegneria dell'Automazione

Università di Bologna, A.A. 2008/2009

Paolo Torroni, Marco Montali

10 Marzo 2009

Esercizio 1 (analisi).

Determinare il valore delle seguenti espressioni in C. Le variabili V, A e B sono di tipo intero, con valore iniziale V=5, A=17, B=34.

```
A<=20||A>=40
!( B=A*2 )
A<=B&&A<=V
A<=( B&&A )<=V
!( A<=B&&A<=V )
!( A>=B )||!( A<=V )
( A++, B=A, V++, A+B+V++ )
```

Esercizio 2 (analisi).

Determinare il valore delle seguenti espressioni in C. Aggiornare il valore delle variabili a seguito della valutazione di ogni espressione.

```
char A=61;
int B=3, C=5;
float R=0.5;

B%=( A / B / C );
R+=A-=B+=C;
R=R-( int )R;
A=B>=( float )3+!!C;
A=A==( R=A );
```

Esercizio 3 (teoria).

Si consideri un'architettura con:

- `unsigned long` a 32 bit.
- `long long` in complemento a due in 64 bit.

Quanti diversi numeri si possono rappresentare con un `unsigned long` e con un `long long`? Qual è il massimo valore rappresentabile nei due casi? Si tratta di rappresentazioni esatte o approssimate?

Esercizio 4 (progetto).

Si consideri un'architettura in cui:

- i `float` hanno una rappresentazione in *floating point* a 32 bit, di cui 23 bit per la mantissa (+1 implicito), 8 bit per l'esponente (in notazione con *bias* pari a $-(2^7 - 1) = -127$, con configurazioni riservate 00000000 e 11111111), e 1 bit per il segno;
- i `double` hanno una rappresentazione in *floating point* a 64 bit, di cui 52 bit per la mantissa (+1 implicito), 11 bit per l'esponente (in notazione con *bias* pari a $-(2^{10} - 1) = -1023$, con 2 configurazioni riservate), e 1 bit per il segno;
- i `long long` sono rappresentati in complemento a due in 64 bit.

Si progetti una serie di esperimenti per evidenziare le seguenti situazioni:

1. l'*overflow*, nei `float` e nei `long long`;
2. la perdita di precisione, senza *overflow*, nella rappresentazione di numeri grandi in `float`;
3. la perdita di precisione nella rappresentazione di numeri piccoli nei `float` (con o senza *underflow*);

Tabella degli operatori in C

Precedenza	Operatori	Associatività
1	`() []`	a sinistra
2	`! ++ --`	a destra
3	`* / %`	a sinistra
4	`+ -`	a sinistra
6	`< <= > >=`	a sinistra
7	`== !=`	a sinistra
11	`&&`	a sinistra
12	`||`	a sinistra
13	`?...:`	a destra
14	`= += -= *=`	a destra
15	`,`	a sinistra

SOLUZIONI

Esercizio 1.

1. `A<=20||A>=40` → `1||0` → `1`

2. `!( B=A*2 )` → `!( B=34 )` → `!34` [`B`←`34`] → `1`

3. `A<=B&&A<=V` → `1&&0` → `0`

4. `A<=( B&&A )<=V` → `17<=1<=5` → `0<=5` → `1`

5. `!( A<=B&&A<=V )` → `!( 0 )` → `1`

6. `!( A>=B )||!( A<=V )`
   → `!( 17>=34 )||!( 17<=5 )`
   → `!( 1 )||!( 0 )`
   → `0||1` → `1`

7. `( A++, B=A, V++, A+B+V++ )`
   → [`A`←`18`] `( B=A, V++, A+B+V++ )`
   → [`B`←`18`] `( V++, A+B+V++ )`
   → [`V`←`6`] `( A+B+V++ )`
   → `( 18+18+6 )` [`V`←`7`] → `42`

Esercizio 2.

1. `B%=( A / B / C )`
   → `B%=( ( int )61 / ( int )3 / C )`
   → `B%=( ( int )20 / ( int )5 )`
   → `B=( int )3 % ( int )4`
   → [`B` ← `3`] `( int )3`;

2. `R+=A-=B+=C`
   → `B=B+5, R+=A-=B`
   → `B=3+5, R+=A-=B`
   → [`B` ← `8`] `R+=A-=8`
   → `A=( int )61-8, R += A`
   → [`A` ← `53`] `R+=53`
   → `R=0.5+( float )53.0, R`
   → [`R` ← `53.5`] `( float )53.5`

3. `R=R-( int )R`
   → `R=( float )53.5-( int )53`
   → `R=( float )53.5-( float )53.0`
   → `R=( float )0.5`
   → [`R` ← `0.5`] `( float )0.5`

4. `A=B>=( float )3+!!C`
   → `A=B>=( float )3.0+!!5`
   → `A=B>=( float )3.0+!0`
   → `A=B>=( float )3.0+( int )1`
   → `A=B>=( float )3.0+( float )1.0`
   → `A=( int )8>=( float )4.0`
   → `A=( float )8.0>=( float )4.0`
   → `A=( int )1`
   → `A=( char )1`
   → [`A` ← `1`] `( char )1`

5. `A=A==( R=A )`
   → `A=A==( R=( char )1 )`
   → `A=A==( R=( float )1.0 )`
   → [`R` ← `1.0`] `A=( char )A==( float )1.0`
   → `A=( float )1.0==( float )1.0`
   → `A=( int )1`
   → `A=( char )1`
   → [`A` ← `1`] `A = ( char )1`

Esercizio 3.

1. `unsigned long`: si possono rappresentare $2^{32} \approx 4 \times 10^9$ (4 miliardi di) numeri interi distinti in modo esatto: tutti quelli compresi nell'intervallo $[0, 2^{32} - 1]$.

2. `long long`: si possono rappresentare $2^{64} \approx 16 \times 10^{18}$ (16 miliardi di miliardi di) numeri interi distinti in modo esatto: tutti quelli compresi nell'intervallo $[-2^{63}, 2^{63} - 1]$.

Esercizio 4.

1. Per generare un *overflow* si può procedere nel modo seguente:

   - si stabilisce il massimo valore rappresentabile con un certo tipo, *max*;
   - si definisce una variabile `x` appartenente a quel tipo;
   - si assegna a `x` un valore vicino a *max*;
   - si moltiplica `x` per 10, in modo da superare *max*.

2. Le variabili `float` consentono di rappresentare in modo esatto tutti gli interi "lunghi" tanto quanto i bit della mantissa. Quindi per osservare la perdita di precisione basta trovare un valore che eccede il massimo rappresentabile con i bit della sola mantissa. Si può procedere nel modo seguente:

   - si stabilisce il massimo valore rappresentabile con la mantissa, come se fosse un intero, *max*;
   - si definisce una variabile `float x`;
   - si assegna a `x` un valore dieci volte superiore rispetto a *max*;
   - si incrementa `x` di una unità e si verifica che il valore rappresentato non cambia rispetto a quello di prima dell'incremento.

3. L'*underflow* si verifica quando si vuole rappresentare un valore minore del minimo rappresentabile. Si può procedere un modo duale rispetto a quanto fatto per l'*overflow*:

- si stabilisce il minimo valore rappresentabile con un certo tipo, *min*;
- si definisce una variabile `x` appartenente a quel tipo;
- si assegna a `x` un valore vicino a *min*;
- si divide `x` per 10.

La perdita di precisione, senza *underflow*, si può osservare assegnando a una variabile `float x` il valore `1.0` e poi sommando a `x` un valore sufficientemente piccolo, ϵ, tale che `float x`$+\epsilon =$`float x`.
Un altro caso di perdita di precisione si osserva con i numeri periodici in binario, che non sono rappresentabili in modo esatto in un numero finito di bit. Sono tutti i numeri la cui parte frazionaria non è riducibile a intera moltiplicandola per potenze di due. Ad esempio, `0.1`. Si può notare come la rappresentazione di `0.1` in `double` ha un errore minore rispetto alla rappresentazione di `0.1` in `float`.

Alcune considerazioni:

- nel caso dei `float`:
  - sono tutti i numeri $\{\pm 1.base \times 2^{exp}\}$;
  - *base* è un intero senza segno a 23 bit, a cui va sommato il bit implicito (1.), quindi è possibile rappresentare interi senza perdita di precisione moltiplicando *base* per $2^2 4$. Il massimo intero rappresentabile senza perdita di precisione quindi è vicino a $2^2 4$. La base comunque ha significato di $1.x$, per cui $\max_{base} < 2$.
  - *exp* è un intero a 8 bit in notazione con *bias* pari a $-(2^7 - 1) = -127$, con configurazioni riservate 00000000 e 11111111, quindi $exp \in [1+bias, 2^8 - 2 + bias]$, ovvero $exp \in [-126, 127]$, quindi $\max_{exp} = 127$;
  - possiamo approssimare *max* come $\max_{base} \times 2^{\max_{exp}} \leq 2 \times 2^{127} = 10^{(\log_{10} 2) \times 128} \approx 10^{38}$;
  - dualmente, possiamo approssimare il minimo numero rappresentabile con 10^{-37}.
- nel caso dei `double`:
  - *base* è un intero senza segno a 53 bit, quindi si possono rappresentare interi senza perdita di precisione più o meno fino a 2^{53}. A ogni modo,$\max_{base} < 2$;
  - *exp* è un intero a 11 bit in notazione con *bias* pari a $-(2^{10} - 1) = -1023$, con configurazioni riservate 00—0 e 11—1, quindi $exp \in [1+bias, 2^1 1 - 2 + bias]$, ovvero $exp \in [-1022, 1023]$, quindi $\max_{exp} = 1023$;
  - possiamo approssimare *max* come $s \times 2^{\max_{exp}} = 2^{1023} = 10^{(\log_{10} 2) \times 1024} \approx 10^{308}$;
  - dualmente, possiamo approssimare il minimo numero rappresentabile con 10^{-307}.
- nel caso dei `long long`:
  - $\max_{\texttt{long long}} = 2^{63} - 1 \approx 8 \times 10^{18}$