Tipi interi – lo standard (1)

Lo standard C lascia al compilatore la decisione di quanti bit assegnare a ogni tipo (char, short int, int, long int).
Quello che è sempre garantito è:

- (signed) char può esprimere valori da -127 a 127
- unsigned char può esprimere valori da 0 a 255
- (signed) short int può esprimere valori da -32767 a 32767
- unsigned short int può esprimere valori da 0 a 65535
- (signed) int e unsigned int seguono le stesse regole di short int, sono stati introdotti per rappresentare la grandezza "naturale" degli interi sulla CPU
- (signed) long int va da -2147483647 a 2147483647
- unsigned long int va da 0 a 4294967295

Tipi interi (2)

Ogni compilatore, rispettando lo standard, può implementare come preferisce le variabili intere. Quello che è valido tipicamente sulle macchine domestiche, e che assumeremo durante il corso, è:

- char è da 8 bit:
  - (signed) char da -128 a 127
  - unsigned char da 0 a 255
- short int è da 16 bit:
  - (signed) short int da -32768 a 32767
  - unsigned short int da 0 a 65535
- int è da 32 bit:
  - (signed) int va da -2147483648 a 2147483647
  - unsigned int va da 0 a 4294967295
- long int è da 64 bit:
  - (signed) long va da -9223372036854775808 a 9223372036854775807
  - unsigned long va da 0 a 18446744073709551615

I valori massimi e minimi di tutti i tipi sono definiti come costanti nell'header limits.h .

Tipi interi – il wrap-around (3)

Tutti i tipi hanno un limite superiore e inferiore. Cosa succede in nei seguenti casi?

```
short int w = 32767;
short int x = -32768;
unsigned short int y = 65535;
unsigned short int z = 0;
w = w + 1; // overflow
x = x - 1; // underflow
y = y + 1; // overflow
z = z - 1; // underflow
```

Tipi interi – il wrap-around (4)

Tutti i tipi interi funzionano come un “anello chiuso”: se si supera il valore massimo si riparte da quello minimo e viceversa, per cui:

```
short int w = 32767;
short int x = -32768;
unsigned short int y = 65535;
unsigned short int z = 0;
w = w + 1; // overflow,  w <- -32768
x = x - 1; // underflow, x <- 32767
y = y + 1; // overflow,  y <- 0
z = z - 1; // underflow, y <- 65535
```

Precisazione: il wraparound è richiesto dallo standard per i tipi unsigned, mentre non è richiesto per i tipi signed, ma in pratica è presente sulla maggior parte delle implementazioni, durante il corso daremo per scontato che avvenga sempre.

Esercizio 2

```
#include <stdio.h>
int main(void)
{
    short int i;
    short int k;

    k = 10000;
    i = 30000 + k;

    return (0);
}
```

1. Copiare, compilare ed eseguire il seguente programma
2. Utilizzando il debug rispondere alle seguenti domande:
   - Quanto valgono *i* e *k* prima degli assegnamenti?
   - Secondo voi, quanto dovrebbe valere *i* dopo l'assegnamento?
   - Quanto vale effettivamente *i* dopo l'assegnamento? Perchè?
3. Modificate il programma, specificando i e k come variabili *unsigned*… cosa cambia? Il comportamento del programma ora è corretto? Perchè?

I tipi reali – float e double (1)

Lo standard prevede due tipi per rappresentare i numeri reali: float e double (“a virgola mobile”). I float sono a “precisione singola”, i double a “precisione doppia” (più precisi). In C qualunque espressione numerica contenente un “.” è considerata un double. Per esplicitare che si desidera che venga considerata come float si aggiunge il suffisso “f”.

```
float a = 5.6f; // questo è un float
double b = 5.6; // questo è un double
```

Un double viene convertito automaticamente in un float in caso di necessità, ma questo *causa una perdita di precisione*.

```
float c = 0.4; // 0.4 è un double che viene convertito
               // in float e assegnato a c
```

I tipi reali – float e double (2)

Il C supporta anche una notazione esponenziale per l'espressione in forma compatta dei valori (valida anche per i tipi interi). La sua sintassi è <coefficiente>e<potenza>, ed equivale a:

<coefficiente> x $10^{<potenza>}$

```
float d = 3.2e4f; // equivale a d = 32000.0f;
double e = 1e3; // equivale a e = 1000.0;
```

Riassumendo, la politica per le costanti numeriche è:
- se la costante contiene un "." è un double
- se al termine del numero con "." c'è una "f" allora la costante è float
- se la costante è in notazione esponenziale la parte prima della "e" è il coefficiente moltiplicativo, la parte dopo è la potenza di 10 per cui moltiplicare il coefficiente.

I tipi reali – float e double (3)

float e double hanno dimensione limitata e usano una rappresentazione binaria. Problemi:
• alcuni numeri che in base 10 hanno rappresentazione finita in base 2 sono periodici (ad esempio 0.1 in base 2 è 0.000110011001 1…)
• presi due numeri reali qualunque, tra essi vi è un intervallo di infiniti numeri reali: una rappresentazione finita non può rappresentarli tutti con precisione.

Lo standard per rappresentare i valori reali sui calcolatori è lo IEEE 754. Lo standard permette di rappresentare:
• infinito
• “Not A Number” (NaN) conseguente ad esempio a divisioni per 0
• approssimazioni di numeri reali

La rappresentazione scelta ha il vantaggio di essere più “densa” vicino allo zero e più “sparsa” mano a mano che i valor rappresentati crescono.

I tipi reali – float e double (4)

Consideriamo la rappresentazione di un float (32 bit). Le componenti di un numero reale sono:
- segno (1 bit)
- esponente (8 bit)
- mantissa o significante (23 bit)

Nei double l'esponente è da 11 bit e la mantissa da 52 bit.
La conversione da rappresentazione IEEE 754 a numero reale è data dalla formula:

$n = (-1)^{segno} \times 2^{esponente} \times (1 + mantissa)$

Ad esempio il numero reale 5.6 è rappresentato come:
- segno: 0
- esponente: 2
- mantissa: 0.39999997615814208984375 (base 2: 0.01100110011001100110011)

Il valore effettivamente rappresentato è quindi:
$(-1)^0 \times 2^2 \times (1 + 0.399999976158\ldots) = 5.599999904632568\ldots$

Esercizio 3

```
#include <stdio.h>
int main(void)
{
    float k;

    k = 5.6F;

    k = k - 5.59F;

    return (0);
}
```

1. Copiare, compilare ed eseguire il seguente programma
2. Utilizzando il debug rispondere alle seguenti domande:
   - Quanto vale *k* prima del primo assegnamento?
   - Quanto vale *k* dopo il primo assegnamento? Quant'e' l'errore di approssimazione
   - Quanto dovrebbe valere, e quanto vale effettivamente *k* dopo il secondo assegnamento? Perché?
3. Modificate il programma, specificando *k* come variabile double... cosa cambia? Quanto vale l'errore di approssimazione?

Espressioni omogenee ed eterogenee (1)

Il C suppone che eseguendo operazioni di qualunque tipo tra tipi omogenei il risultato ottenuto debba essere dello stesso tipo. Ad esempio se si divide un int per un altro int, il risultato è int – per cui viene persa la parte decimale:

```
int a = 7 / 3; // a vale 2
```

Se le operazioni vengono effettuate tra tipi di dimensioni differenti, le costanti e le variabili vengono promosse al tipo di dimensione superiore, ad esempio:

```
short int b = 42;
int c = 2;
long int d = 1;
long int e;
e = (b + c) * d;
```

- **`b + c`**: b è promosso a int e l'intera espressione è valutata come int;
- **`(b + c)*d`**: l'int appena ottenuto viene promosso a long per effettuare l'operazione con d

Se il risultato di un'operazione viene assegnato a una variabile di dimensione troppo piccola per contenerlo, perdiamo dei bit di informazione, e nella variabile di destinazione troveremo bit "casuali" (avremo i bit meno significativi del valore che volevamo memorizzare).

Espressioni omogenee ed eterogenee (2)

Lo stesso tipo di promozione si ha anche operando con float e double: se un float opera con un double viene promosso a sua volta a double.

Se un tipo intero opera con tipi reali, viene promosso al tipo reale.

```
int c = 5;
int d = c/2; // d vale 2
float e = c/2.0F; // e vale 2.5F
```

Cosa succede assegnando un valore di tipo float o double a un tipo intero? C'è il troncamento: tutte le cifre decimali del numero reale vengono scartate e alla variabile intera verrà assegnata la parte intera del numero originario:

```
float pigreco = 3.141592f;
int x = pigreco; // x vale 3
```

A questo si aggiungono comunque le incompatibilità di segno che causano wrap-around!

```
float a = -3.141592f;
unsigned short int y = a; // y vale 65533;
```

Esercizio 4

```
#include <stdio.h>
int main(void)
{
    int i, k;
    float j;

    i = 20;
    k = i % 3;
    i = i / 3;

    k = i / 4.0F;
    j = i / 4.0F;

    return (0);
}
```

1. Copiare, compilare ed eseguire il seguente programma

2. Utilizzando il debug e le finestre di “watch”/”locals”, rispondere alle seguenti domande:
   - Quanto valgono i e k dopo il primo blocco di assegnamenti?
   - Quanto valgono k e j dopo il secondo blocco di assegnamenti?
   - Se k e j al termine del programma hanno valori diversi, perchè?

3. In fase di compilazione il programma potrebbe aver generato dei warnings…
Leggete i warning e spiegate il loro significato
Correggete il codice al fine di far scomparire i warning