Tipi di dati scalari (casting e puntatori)

Alessandra Giordani
agiordani@disi.unitn.it
Lunedì 10 maggio 2010
http://disi.unitn.it/~agiordani/

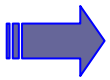
I tipi di dati scalari

- I tipi aritmetici, i tipi enumerativi ed i puntatori vengono detti tipi scalari, poiché i valori che li compongono sono distribuiti su una *scala lineare*, su cui si può stabilire una relazione di ordine totale
- La dichiarazione fornisce al compilatore le informazioni relative al numero di byte da allocare e alle modalità di interpretazione di tali byte
- Le parole chiave char, int, float, double, ed enum descrivono i tipi base; short, long, signed, unsigned sono i qualificatori che modificano i tipi base

Qualificatori short/long

- Al tipo int possono essere assegnate dimensioni diverse su architetture distinte (tipicamente 4 o 8 byte)
- Il tipo int rappresenta il formato “naturale” per il calcolatore, ossia il numero di bit che la CPU manipola normalmente in una singola istruzione
- Supponiamo che int corrisponda a celle di memoria di 4 byte:
  - Il tipo short int corrisponde generalmente a 2 byte
  - Il tipo long int a 4/8 byte
- Nelle dichiarazioni di interi short/long la parola int può essere omessa

```
short int j;
long int k;
```



```
short j;
long k;
```

Qualificatori unsigned/signed

- Si possono individuare casi in cui una variabile può assumere solo valori positivi (ad es., i contatori)
- Il bit più significativo non viene interpretato come bit di segno
- Esempio: una variabile short int può contenere i numeri interi compresi fra –32768 e 32767, mentre una variabile dichiarata unsigned short int può contenere valori da 0 a 65535

unsigned (int) p;

- Lo specificatore signed consente di definire esplicitamente una variabile che può assumere valori sia positivi che negativi
- Normalmente signed è superfluo, perché i numeri interi sono con segno *per default*

I tipi interi

Tipo	Byte	Rango
int	4	da -2^{31} a $2^{31}-1$
short int	2	da -2^{15} a $2^{15}-1$
long int	4	da -2^{31} a $2^{31}-1$
	8	da -2^{63} a $2^{63}-1$
unsigned int	4	da 0 a $2^{32}-1$
unsigned short int	2	da 0 a $2^{16}-1$
unsigned long int	4	da 0 a $2^{32}-1$
signed char	1	da -2^{7} a $2^{7}-1$
unsigned char	1	da 0 a $2^{8}-1$

Dimensione e rango dei valori dei tipi interi sulla macchina di riferimento

Caratteri e interi – 1

- La maggior parte dei linguaggi distingue i caratteri dai dati numerici: 5 è un numero mentre 'A' è un carattere
- In C, la differenza tra carattere e numero è sfumata: il tipo di dati char è un valore intero rappresentato con un byte, che può essere utilizzato per memorizzare sia caratteri che interi
- Per esempio, dopo la dichiarazione

  ```
  char c;
  ```

  i seguenti assegnamenti sono corretti ed equivalenti:

  ```
  c='A';
  c=65;
  ```

  In entrambi i casi, viene assegnato alla variabile c il valore 65, corrispondente al codice ASCII della lettera A

Caratteri e interi – 2

- Le costanti di tipo carattere sono racchiuse tra apici singoli
- Esempio: Leggere un carattere da terminale e visualizzarne il codice numerico

```c
/* Stampa del codice numerico di un carattere */
#include<stdio.h>

int main()
{
    char ch;

    printf("Digitare un carattere: ");
    scanf("%c", &ch);
    printf("Il codice numerico corrispondente e' %d\n", ch);
    return 0;
}
```

Caratteri e interi – 3

- Dato che in C i caratteri sono trattati come interi, su di essi è possibile effettuare operazioni aritmetiche

  int j = 'A'+'B';

  j conterrà il valore 131, somma dei codici ASCII 65 e 66

Esempio: Scrivere una funzione che converte un carattere da maiuscolo a minuscolo

Funziona per la codifica ASCII

```
#include<stdio.h>

char to_lower(char ch)
{
    return ch+32;
}

int main()
{
    char ch;

    printf("Digitare un carattere MAIUSCOLO [A-Z]: ");
    scanf("%c", &ch);
    printf("Ecco lo stesso carattere minuscolo: %c\n", to_lower(ch));
    return 0;
}
```

Le tipologie di costanti intere – 1

- Oltre alle costanti decimali, il C permette la definizione di costanti ottali ed esadecimali
- Le costanti ottali vengono definite anteponendo al valore ottale la cifra 0
- Le costanti esadecimali vengono definite anteponendo la cifra 0 e x o X

Decimale	Ottale	Esadecimale
3	03	0x3
8	010	0X8
15	017	0xF
16	020	0x10
21	025	0x15
–87	–0127	–0x57
187	0273	0xBB
255	0377	0Xff

Le tipologie di costanti intere – 2

- Esempio: Leggere un numero esadecimale da terminale e stampare gli equivalenti ottale e decimale

```
/* Stampa gli equivalenti ottale e decimale
 * di una costante esadecimale
 */
#include<stdio.h>

int main()
{
    int num; char a;

    printf("Digitare una costante esadecimale: ");
    scanf("%x", &num);
    printf("L'equivalente decimale di %x e' %d\n", num, num);
    printf("L'equivalente ottale di %x e' %o\n", num, num);
    return 0;
}
```

Le combinazioni di tipi – 1

- Nelle espressioni, il C ammette la combinazione di tipi aritmetici:

  num=3*2.1;

  l'espressione è la combinazione di un int ed un double; inoltre num potrebbe essere di qualunque tipo scalare, eccetto un puntatore
- Per associare un significato alle espressioni contenenti dati di tipi diversi, il C effettua automaticamente un insieme di *conversioni implicite*:

  3.0+1/2

  verrebbe valutata 3.0 anziché 3.5, dato che la divisione viene effettuata in aritmetica intera

Le combinazioni di tipi – 2

- Le conversioni implicite vengono effettuate in quattro circostanze:
  - Conversioni di assegnamento — nelle istruzioni di assegnamento, il valore dell'espressione a destra viene convertito nel tipo della variabile di sinistra
  - Conversioni ad ampiezza intera — quando un char od uno short int appaiono in un'espressione vengono convertiti in int; unsigned char ed unsigned short vengono convertiti in int, se int può rappresentare il loro valore, altrimenti sono convertiti in unsigned int
  - In un espressione aritmetica, gli oggetti sono convertiti per adeguarsi alle regole di conversione dell'operatore
  - Può essere necessario convertire gli argomenti di funzione

Le combinazioni di tipi – 3

- Per le conversioni di assegnamento, sia j un int e si consideri…

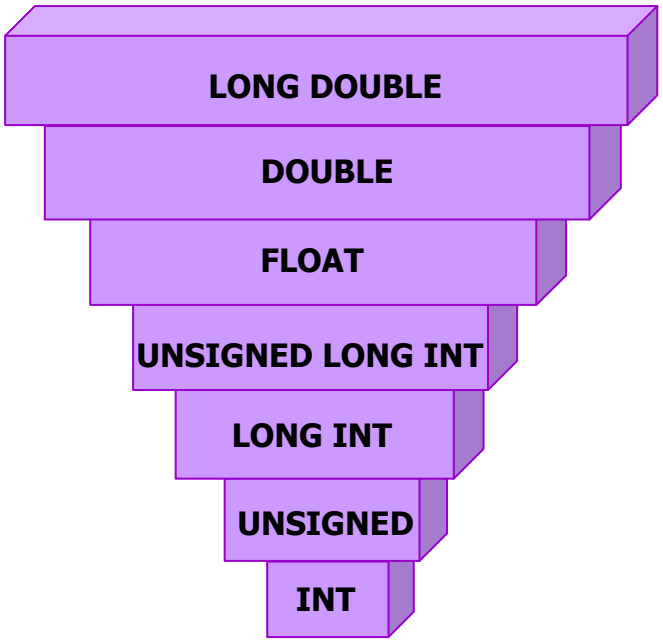
  j=2.6;

  Prima di assegnare la costante di tipo double, il compilatore la converte in int, per cui j assume il valore intero 2 (agisce per troncamento, non per arrotondamento)

- La conversione ad ampiezza intera o *promozione ad intero*, avviene generalmente in modo trasparente

Le combinazioni di tipi – 4

- L'analisi di un'espressione da parte del compilatore ne comporta la suddivisione in sottoespressioni; gli operatori binari impongono operandi dello stesso tipo: l'operando il cui tipo è "gerarchicamente inferiore" viene convertito al tipo superiore:



Esempio**:** La somma fra un int e un double (1+2.5) viene valutata come (1.0+2.5)

La combinazione di floating-point

- L'uso congiunto di float, double e long double nella stessa espressione fa sì che il compilatore, dopo aver diviso l'espressione in sottoespressioni, ampli l'oggetto più corto di ogni coppia associata ad un operatore binario
- In molte architetture, i calcoli effettuati sui float sono molto più veloci che quelli relativi a double e long double…
  - I tipi di numeri più ampi dovrebbero essere impiegati solo quando occorre una grande precisione o occorre memorizzare numeri molto grandi
- Possono esserci problemi quando si effettuano conversioni da un tipo più ampio ad uno meno ampio
  - Perdita di precisione
  - Overflow

Le conversioni di tipo esplicite: cast

- In C, è possibile convertire esplicitamente un valore in un tipo diverso effettuando un cast
- Per realizzare una conversione di tipo esplicita di un'espressione, si pone tra parentesi tonde, prima dell'espressione, il tipo in cui si desidera convertire il risultato

```
#include<stdio.h>

int main()
{
    int j, k;
    float f;

    printf("Inserire due valori j,k. Calcorero j/k con e senza casting\n");
    scanf("%d %d", &j, &k);
    f=j/k;
    printf("valore di f=%d/%d senza casting= %f\n",j,k,f);
    f=(float)j/k; //conversione esplicita
    printf("valore di f=%d/%d con casting= %f\n",j,k,f);
    f=(j*1.0)/k; //conversione implicita
    printf("valore di f=%d/%d con 1.0= %f\n",j,k,f);
    return 0;
}
```

Puntatori - 1

- Un puntatore é una variabile che rappresenta un indirizzo di memoria
- Usati principalmente per: memorizzare indirizzi di variabili, passaggio per riferimento, manipolazione di array
- I puntatori in C sono "tipati": portano con sé informazione relativa al tipo di dato a cui puntano
- Dichiarazione di un puntatore: operatore unario "*" (solo in dichiarazione!!)

```
int a, b, c; /* dichiaro tre variabili intere */
int * x;     /* dichiaro un puntatore a intero */
double * y;  /* dichiaro un puntatore a double */
```

- Una variabile puntatore occupa in memoria un numero di byte sufficiente a rappresentare tutti gli indirizzi di memoria indirizzabili dal bus $\Rightarrow$ 4 byte in una architettura a 32 bit
- Lo spazio occupato da un puntatore non dipende dal tipo!

Puntatori - 2

- N.B.: * nella dichiarazione é associato a destra (al nome di variabile) e non a sinistra (al tipo). Se voglio dichiarare due puntatori ad intero sulla stessa riga devo scrivere

```
int * j, * k;
```

- Se invece scrivessi:

```
int * j,k;
```

  dichiarerei:

  - Un puntatore ad intero (j)
  - Una variabile intera (k)

- Domanda: quale é l'effetto della seguente istruzione?

```
double a, * b, c, d;
```

Manipolazione dei puntatori

- Operatore indirizzo di (unario): &
  - Variabile ⇒ Puntatore
  - Resituisce l'indirizzo (reference) di una variabile (e ci permette di salvarla in un puntatore!)
  - Da non confondere con l'AND logico (binario, &&) e l'AND bit a bit (binario, &)
- Operatore di dereferenziazione (unario, nelle espressioni): *
  - Puntatore ⇒ Valore
  - Restituisce il valore della variabile che si trova ad un certo indirizzo (e ci permette di manipolare il valore di una variabile di cui conosciamo l'indirizzo!)
  - Da non confondere con l'operatore di moltiplicazione (binario) o con l'operatore * nella dichiarazione di una variabile di tipo puntatore

Attenzione:

- La dichiarazione:

```
int * a;
```

- Significa: dichiaro un puntatore ad intero e lo chiamo a
- Il nome del puntatore é solo a, senza * !!!

- a rappresenta un indirizzo

- *a é il valore memorizzato all'indirizzo a

Esempio:

```
int a = 100, b = 50, *p;

p = &a;          // ora p punta ad a

int c = *p;      // assegno a c il valore puntato da p: 100

p = &b;          // ora p punta a b

c = *p + 10;     // assegno a c il valore puntato da p
                 // aumentato di 10, cioé 60

*p = *p + 5;     // equivalente a: b = b + 5, b vale 55

int * q = &a;

int d = *p * *q; // cioé: d = (*p) * (*q)
                 // d value 55*100 = 5500
```

Puntatori e printf

- La funzione printf permette di stampare un indirizzo mediante l'indicatore di formato %p:

```
int a = 100;
printf("a: %d, indirizzo: %p\n", a, &a);

int *b = &a;
printf("b: %p, indirizzo: %p\n", b, &b);
return 0;
```

- L'indirizzo viene stampato in formato esadecimale

- Provate a compilare l'esempio di prima stampando i valori di tutte le variabili!

Passaggio di parametri per valore

- I puntatori sono alla base del meccanismo di passaggio per riferimento (passing by reference)
- In C, le funzioni lavorano su copie locali dei parametri attuali che gli vengono passati
- Ne segue che una funzione non puó modificare il contenuto delle variabili che gli vengono passate

```
void myfunc(int a) { a += 5};

int main() {
    int value = 10;
    myfunc(value);
    printf("%d\n", value); //stampa 10
}
```

Passaggio di parametri per riferimento

- Passando puntatori (references) anziché valori (values), possiamo scrivere funzioni che modificano il valore delle variabili che gli vengono passate

```
void myfunc(int* a) { *a += 5};

int main() {
    int value = 10;
    myfunc(&value);
    printf("%d\n", value); //stampa 15
}
```

- Se una funzione deve modificare il valore di un parametro, il parametro deve essere passato per riferimento

Esempio di passaggio parametri per valore e/o riferimento 1

- Arrotondare una variabile double passata per valore ad un intero restituito come valore di ritorno:

```
int round_return_by_value(double value)
{
    if ( (value - floor(value)) >= 0.5 ) {
        return ceil(value);
    }
    return floor(value);
}
```

Esempio di passaggio parametri per valore e/o riferimento 2

- Arrotondare una variabile double passata per valore ad un intero passato per riferimento come parametro:

```
void round_return_by_reference(double value, int* rounded)
{
    if ( (value - floor(value)) >= 0.5 ) {
        *rounded = ceil(value);
    } else {
        *rounded = floor(value);
    }
}
```

Esempio di passaggio parametri per valore e/o riferimento 3

- Arrotondare una variabile double passata per riferimento modificando il contenuto puntato da essa:

```
void round_in_place(double* value)
{
    if ( (*value - floor(*value)) >= 0.5 ) {
        *value = ceil(*value);
    } else {
        *value = floor(*value);
    }
}
```

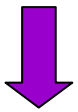
Esempio di passaggio parametri per valore e/o riferimento 4

```
int main()
{
    double d, *pd=&d;
    int i, *pi=&i, tmp;

    printf("Inserire un numero decimale: ");
    scanf("%lf", pd);
    //scanf("%lf", &d);

    i=round_return_by_value(*pd);
    //i=round_return_by_value(d);
    printf("Arrotondamento per valore= %d\n",i);

    round_return_by_reference(*pd, &tmp);
    //round_return_by_reference(d, &tmp);
    printf("Arrotondamento per referenza= %d\n",tmp);

    round_in_place(pd);
    //round_in_place(&d);
    printf("Arrotondamento al volo= %f\n",*pd);

    return 0;
}
```

La dichiarazione di tipo: typedef

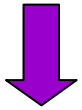
- La dichiarazione di tipo deve apparire in un programma prima che il tipo venga adoperato per la dichiarazione di variabili
- Le dichiarazioni di tipo sono particolarmente utili nella definizione di tipi composti
- Avvertenza: typedef e #define non sono equivalenti…

```
#define PT_TO_INT  int *

PT_TO_INT p1, p2;
```



```
int  *p1, p2;
```

```
typedef  int *  PT_TO_INT;

PT_TO_INT p1, p2;
```



```
int  *p1, *p2;
```