Programmazione (imperativa)

Corso di Laurea in Informatica

Roberto Cordone

DI - Università degli Studi di Milano

Lezioni:	Lunedì 12.00 - 13.00 e 14.00 - 16.00	Mercoledì 14.00 - 17.00
Laboratorio:	Giovedì 12.00 - 13.00 e 14.00 - 17.00	
Ricevimento:	su appuntamento	
Tel.:	02 503 16235	
E-mail:	roberto.cordone@unimi.it	
Web page:	http://homes.di.unimi.it/~cordone/courses/2015-prog/2015-prog.html	

Tipi di dato

Ogni variabile (e ogni costante) è associata a un tipo, che indica

- l'insieme dei valori che può assumere (se variabile)
- il formato binario del valore
- lo spazio occupato in memoria dal valore
- le operazioni che si possono compiere con essa e come compierle

Per ogni variabile occorre una dichiarazione all'inizio del blocco

tipo nome ;

Esempi: `int i, j;` `char c;` `float r, s;`

La dichiarazione è un'istruzione (termina con ;): indica al processore di

- riservare al dato lo spazio in memoria corretto
- aggiungere il nome e l'indirizzo del dato alla tabella dei simboli

Tipi di dato

I tipi di dato si distinguono in

- tipi semplici: rappresentano informazioni composte da un solo valore (un voto, una temperatura, un prezzo, un mese)
- tipi strutturati: rappresentano informazioni composte dall'insieme di più valori concettualmente legati (i risultati di un esame, le misure di una stazione meteorologica, un listino prezzi, una data)

Inoltre, dato un linguaggio di programmazione, vi sono

- tipi predefiniti forniti dal linguaggio stesso
- tipi definiti dall'utente con opportuni costrutti

Tipi predefiniti

Il C fornisce

- tre tipi di numeri interi: `short`, `int` e `long`
- tre tipi di numeri naturali: `unsigned short`, `unsigned int` e `unsigned long`
- tre tipi di numeri reali: `float`, `double` e `long double`
- tre tipi di caratteri: `char`, `signed char` e `unsigned char` (ignoreremo gli ultimi due)

Tutti questi tipi differiscono per l'intervallo di definizione

Si noti che tutti descrivono insiemi finiti

Numeri interi

I numeri interi assumono valori interi compresi fra -2^{n-1} e $2^{n-1}-1$

- n è il numero di bit occupati in memoria
- n va crescendo da `short` a `int` a `long`
- n dipende dal compilatore e dal processore
- costanti simboliche per gli estremi sono definite in `limits.h`

I valori tipici per n sono potenze di 2; attualmente il `gcc` pone

- per il tipo `short`, $n = 16$: da $-32\,768$ a $32\,767$
- per il tipo `int`, $n = 32$: da $-2\,147\,483\,648$ a $2\,147\,483\,647$
- per il tipo `long`, $n = 32$: da $-2\,147\,483\,648$ a $2\,147\,483\,647$

Numeri naturali

Assumono valori compresi fra 0 e $2^n - 1$

- n è il numero di bit occupati in memoria
- n va crescendo da `unsigned short` a `unsigned int` a `unsigned long`
- n è identico al valore del corrispondente tipo intero
- n dipende dal compilatore e dal processore
- costanti simboliche per gli estremi sono definite in `limits.h`

Attualmente il `gcc` pone

- per il tipo `unsigned short`, $n = 16$: da 0 a 65 535
- per il tipo `unsigned int`, $n = 32$: da 0 a 4 294 967 295
- per il tipo `unsigned long`, $n = 32$: da 0 a 4 294 967 295

Numeri reali

Non sono veramente numeri reali, ma razionali, perché occupano uno spazio finito; quindi hanno

- un intervallo di definizione
- un margine di precisione: un intervallo reale è rappresentato da un singolo valore

Intervallo e margine dipendono dal numero di bit n usato

- n va crescendo da `float` a `double` a `long double`
- n dipende dal compilatore e dal processore
- costanti simboliche per intervallo e margine sono definite in `float.h`

Caratteri

L'insieme dei caratteri disponibili dipende dal processore

- lo standard ASCII ne comprende 128
- lo standard Latin-1 li estende a 256 (cioè 2^n con $n = 8$)

I caratteri sono un insieme ordinato

- costanti simboliche per gli estremi sono definite in `limits.h`
- si possono applicare gli operatori relazionali <, <=, ecc...
- si possono incrementare e decrementare e
  si possono sommare e sottrarre numeri interi ai caratteri:

  Esempi:

  `i = 'f'; i++;` (*ora* `i` *vale* `'g'`)

  `i = 'f'; i -= 4;` (*ora* `i` *vale* `'b'`)

Costanti (1)

Il tipo di una costante è definito implicitamente dalla sua forma

- è composta solo da cifre: numero intero
- comincia con `0x` seguita da cifre o lettere comprese fra `A` e `F`: numero intero esadecimale
- comincia con una cifra e contiene un punto: numero reale
- comincia con una cifra e contiene una `E`: numero reale in notazione esponenziale

  $$n\texttt{E}e \quad \text{significa} \quad n * 10^e$$

  Esempi: `5.734E+2` vale `573.4`
  `57.34E-2` vale `0.5734`
- è racchiusa fra apici: carattere

Costanti (2)

È bene esplicitare il tipo delle costanti a vantaggio del lettore umano

- aggiungendo `L` in coda alle costanti di tipo `long`:
  `100 000L` anziché `100 000`
- aggiungendo `U` in coda alle costanti di tipo `unsigned`:
  `100 000U` o `100 000UL` anziché `100 000`
- aggiungendo un punto (`.`) alle costanti di tipo `double`:
  `100 000.0` anziché `100 000` (per motivi storici, le costanti reali sono `double`, se non altrimenti indicato)
- aggiungendo una `f` alle costanti di tipo `float`:
  `100 000f` anziché `100 000.0`

Questo consente anche di forzare un tipo diverso da quello assegnato implicitamente: `1L` diventa di tipo `long`, mentre `1` sarebbe `int`

Sequenze di *escape*

Vi sono caratteri che non si possono stampare direttamente

- perché non hanno una forma visibile (a capo,...)
- perché sono usati per altri scopi (virgolette, apici,...)

Si rappresentano con un carattere preceduto da \

Carattere	Sequenza
A capo	\n
Tabulazione	\t
Backspace	\b

Carattere	Sequenza
Virgolette	\"
Apice	\'
Barra	\\

Situazioni di errore

Overflow è l'uscita del valore di un'espressione dall'intervallo associato al tipo; in caso di *overflow*, l'esecuzione continua:

- con valori indefiniti per i tipi interi e reali
- in aritmetica modulo 2^n per i tipi naturali

Si possono avere risultati sbagliati senza segnalazioni di errore!

A causa del margine di precisione limitato

- due numeri reali diversi possono risultare identici
- due numeri reali identici possono risultare diversi
  (ad es., `(a + b) != (b + a)`)

Quindi, non si deve scrivere `if (r == s)`,
ma `if ((-EPSILON <= r - s) && (r - s <= EPSILON))`

(*avendo definito un opportuno valore di* `EPSILON`)

Conversioni di tipo

Ogni operazione richiede che gli operandi siano tutti dello stesso tipo

Però è utile avere espressioni miste, cioè con operandi di tipo diverso

Il C le gestisce con

- regole implicite di conversione
- operatori espliciti di conversione

La conversione implicita avviene quando

1. un'espressione composta ha operandi di tipo diverso
2. un assegnamento ha il lato destro di tipo diverso dal lato sinistro
3. una chiamata a funzione ha argomenti di tipo diverso dai parametri corrispondenti
4. un'istruzione `return` è seguita da un'espressione di tipo diverso dal risultato della funzione

Conversioni di espressioni composte

Quando un'espressione composta ha operandi misti:

1. gli operandi di tipo `char` si convertono in tipo `int`
2. se gli operandi sono tutti interi o naturali oppure tutti reali, si converte il tipo più "limitato" nel tipo più "ampio"

   `int` → `unsigned int` → `long` → `unsigned long`

   `float` → `double` → `long double`

3. se almeno un operando è reale, si convertono tutti in reali

Conversioni negli assegnamenti

variabile = *espressione*;

Si converte il valore dell'espressione nel tipo della variabile

- se il tipo della variabile è più "ampio": si converte banalmente
- se il tipo della variabile è più "limitato":
  - valore intero e variabile intera di tipo inferiore o carattere: se il valore rispetta l'intervallo, si converte; altrimenti, diventa indefinito
  - valore reale e variabile reale di tipo inferiore: se il valore rispetta l'intervallo, si approssima; altrimenti, diventa indefinito
  - valore reale e variabile intera: si tronca il valore alla parte intera

Osservazioni

- si possono rappresentare piccoli interi con caratteri per risparmiare memoria (in casi critici!)
- è pericoloso mescolare interi e naturali: il risultato potrebbe essere di entrambi i tipi
- bisogna badare bene alle precedenze fra operatori: il risultato può essere diverso da quello atteso

Conversioni esplicite (1)

(*tipo*) *espressione* ;

L'operatore di *cast* () converte il valore dell'espressione nel tipo

- per superare le regole di conversione implicite
- per esplicitare le conversioni non banali

È un operatore unario ad alta priorità

Dati: `int a, b; double r;`

`r = a/b;` calcola la divisione troncata e la converte in `double`

`r = (double) a/b;` converte prima `a`, poi `b` e calcola la divisione esatta

`r = a/(double) b;` converte prima `b`, poi `a` e calcola la divisione esatta

`r = (double) a/(double) b;` converte prima `a`, poi `b` e calcola la divisione esatta

Conversioni esplicite (2)

Dati: `long i; short j;`

- `i = j * j;`
  1. calcola il prodotto `short` in aritmetica modulare
  2. lo converte in `long` e lo assegna ad `i`
- `i = (long) j * j;`
  1. converte il primo `j` in `long` a causa del cast
  2. converte il secondo `j` in `long` implicitamente
  3. calcola il prodotto `long` e lo assegna ad `i`
- `i = (long) (j * j);`
  1. calcola il prodotto `short` in aritmetica modulare
  2. lo converte in `long` a causa del cast
  3. lo assegna ad `i`

Tipi definiti da utente (1)

`typedef` *tipo_esistente tipo_nuovo*`;`

In C si può dare un nuovo nome ai tipi di dato esistenti
(predefiniti o definiti in precedenza dall'utente)

- per rendere più chiaro il significato del dato
- per facilitare modifiche successive al codice

```
typedef int anno;
anno a; a = 2010;
```

I dati di questo tipo...
...sono interi che rappresentano anni

```
typedef int boolean;
boolean b; b = TRUE;
```

...assumeranno solo valori logici

```
typedef anno annobisestile;
annobisestile ab; ab = 2000;
```

...rappresentano anni bisestili

Tipi definiti da utente (2)

La definizione di tipo

- se inserita nella parte dichiarativa di un blocco
  vale solo nel blocco cui appartiene
- se inserita fra le direttive e i prototipi vale per l'intero file
  (dichiarazioni globali)

La definizione di tipo termina con ; perché è un'istruzione:
aggiunge un nuovo nome alla tabella dei tipi di dato

Tipi enumerativi

Tipo enumerativo è un tipo definito da utente con un numero finito di valori, esplicitamente enumerati nella dichiarazione

Tipicamente, serve a definire valori simbolici

Una variabile di tipo enumerativo si dichiara specificando

- l'elenco dei valori possibili
- il nome della variabile

`enum` { *valore1*, *valore2*, ... } *variabile*;

Esempio: `enum { PICCHE, CUORI, QUADRI, FIORI } s;`

È la solita struttura "*tipo variabile*;", ma il *tipo* è composto da più parole

Enumerazioni e interi (1)

Il C tratta le espressioni di tipo enumerativo come interi,
cioè assegna agli elementi del tipo i valori 0, 1, ...

```
enum { PICCHE, CUORI, QUADRI, FIORI } s1, s2;
```

equivale a

```
#define PICCHE 0
#define CUORI 1
#define QUADRI 2
#define FIORI 3
int s1, s2;
```

salvo che

- il precompilatore non sostituisce le costanti coi loro valori
- i valori simbolici sono locali al blocco che li dichiara

Enumerazioni e interi (2)

Si possono assegnare esplicitamente i valori alle costanti

`enum` { *costante1* = *valore1*, *costante2* = *valore2*, ... } *variabile*;

Ad esempio: `enum { TRUE = 1, FALSE = 0 } b;`

- enumerativi e interi si convertono banalmente gli uni negli altri
- si possono applicare gli operatori interi ai tipi enumerativi
- si possono usare i valori enumerativi come indici
  - nei cicli
  - nei vettori
  - nel costrutto `switch`

  Ovviamente, questo comporta un rischio di *overflow* non segnalati

Dichiarazione di tipi enumerativi

Ripetere una dichiarazione di tipo enumerativo per ogni variabile

```
enum { FALSE, TRUE } b1;
enum { FALSE, TRUE } b2;
...
```

è poco chiaro e poco compatto

Vogliamo dare un nome simbolico al tipo (per es., `boolean`) e separare

- la dichiarazione della variabile (`b1`, `b2`, ...)
- dalla dichiarazione del tipo (`boolean`)

Si può fare in due modi:

1. con l'istruzione `typedef`

   ```
   typedef enum { FALSE, TRUE } boolean;
   boolean b;
   ```

2. con i tag di enumerazione

   ```
   enum boolean { FALSE, TRUE };
   enum boolean b;
   ```

L'operatore `sizeof`

`sizeof(` *tipo* `);` `sizeof(` *espressione* `);`

Il valore dell'espressione `sizeof(...)` è lo spazio occupato

- da oggetti del tipo *tipo*
- dall'espressione *espressione*

Il valore è di tipo `size_t` (numero naturale)

Di solito viene usato solo per allocare memoria dinamica
(tema di una lezione successiva)