TIPI DI DATO

Un **tipo di dato** T è definito come:

- un **dominio di valori, D**
- un **insieme di funzioni $F_1,..,F_n$** sul dominio D
- un **insieme di predicati $P_1,..,P_m$** sul dominio D

$T = \{ D, \{F_1,...,F_n\}, \{P_1,...,P_m\} \}$

COMPATIBILITA' DI TIPO

- **Consiste nella possibilità di usare, *entro certi limiti*, oggetti di un tipo *al posto di oggetti di un altro tipo.***
- **Un tipo T1 è *compatibile* con un tipo T2 se il dominio D1 di T1 *è contenuto* nel dominio D2 di T2.**
  - **`int` è compatibile con `float` perché $Z \subset R$**
  - **ma `float` *non è compatibile* con `int`**

COMPATIBILITA' DI TIPO

- **Se T1 è un tipo compatibile con T2, un operatore definito per T2 può essere utilizzato *anche con parametri di tipo T1***

In particolare se Op è definito per T2 come:

Op: $D2 \times D2 \rightarrow D2$

allora può essere utilizzato anche per T1, come:

Op: $\mathbf{D1} \times D2 \rightarrow D2$

Op: $D2 \times \mathbf{D1} \rightarrow D2$

Gli operandi di tipo T1 sono *convertiti automaticamente* nel tipo T2 - **il risultato è sempre di tipo T2.**

COMPATIBILITA' DI TIPO

- **Se T1 è un tipo compatibile con T2, un operatore definito per T2 può essere anche utilizzato con parametri di tipo T1**

In particolare se Op è definito per T2 come:

Op: $D2 \times D2 \rightarrow D2$

allora può essere utilizzato anche per T1, come:

Op: $\mathbf{D1} \times D2 \rightarrow D2$

Op: $D2 \times \mathbf{D1} \rightarrow D2$

Gli operandi di tipo T1 sono *convertiti automaticamente* nel tipo T2 - il risultato è sempre di tipo T2.

COMPATIBILITA' DI TIPO - NOTA

- 3 / 4.2 è una divisione *fra reali,* in cui il primo operando è convertito automaticamente da **`int`** a **`double`**

- 3 % 4.2 è una operazione *non ammissibile*, perché 4.2 non può essere convertito in **`int`**

CONVERSIONI IMPLICITE

1) Ogni variabile di tipo **char** o **short** (eventualmente con qualifica **signed** o **unsigned**) viene convertita in **int**

2) Se dopo il passo 1 l'espressione è ancora eterogenea, si converte temporaneamente l'operando di *tipo inferiore* al *tipo superiore:*

   char → *int* → *long int* → *float* → *double* → *long double*

3) A questo punto l'espressione è omogenea e viene invocata l'operazione opportuna. Il risultato è dello stesso tipo.

CONVERSIONI IMPLICITE

Esempio

```
int x;
char y;
double r;
```

L'espressione:

```
(x+y) / r
```

produce un risultato di tipo `double`.

COMPATIBILITA' IN ASSEGNAMENTO

- In un assegnamento, l'identificatore di variabile e l'espressione devono essere *dello stesso tipo*
- Se così non è, scatta una *conversione implicita*

Esempio

```
int x=5; char y='a'; double r=3.1415;
x = y;    /* conversione da char ad int */
x = y+x;
r = y;    /* char --> int --> double    */
```

COMPATIBILITA' IN ASSEGNAMENTO

- In generale, sono automatiche le conversioni di tipo che non provocano perdita d'informazione
- Espressioni che *possono* provocare perdita di informazioni non sono però illegali

Esempio

```
int x=5; float f=2.71F;; double d=3.1415;
f = f+i;  /* int convertito in float */
i = d/f;  /* double convertito in int !*/
f = d;  /* arrotondamento o troncamento */
```

Possibile Warning: *conversion may lose significant digits*

CAST

- In qualunque espressione è possibile **forzare una particolare conversione** utilizzando l'***operatore di cast***

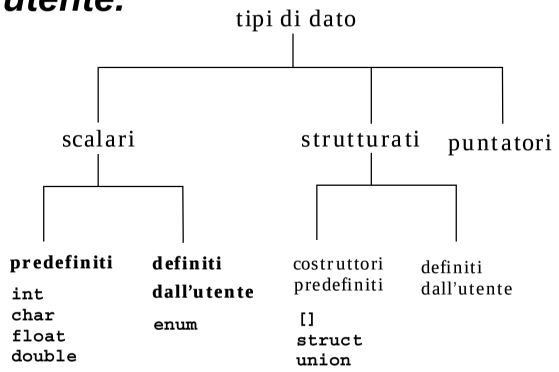
(<tipo>) <espressione>

Esempi

```
int i=5; long double x=7.77; double y=7.1;
i = (int) sqrt(384);
x = (long double) y*y;
i = (int) x % (int)y;
```

TIPI DI DATO

I tipo di dato, *scalari* o *strutturati,* si differenziano in *predefiniti e definiti dall'utente.*



TIPI DI DATO

- **Ogni elaboratore è *intrinsecamente capace* di trattare domini di dati di *tipi primitivi***
  - ***numeri naturali, interi, reali***
  - ***caratteri* e *stringhe di caratteri***

  e quasi sempre anche collezioni di oggetti, mediante la definizione di *tipi strutturati*
  - ***array, strutture***
- **Altri tipi possono essere *definiti dall'utente.***

TIPI DEFINITI DALL'UTENTE

- In C, l'utente può introdurre *nuovi tipi* tramite una *definizione di tipo*
- La definizione associa a un identificatore (*nome del tipo*) un tipo di dato
  - aumenta la leggibilità del programma
  - consente di ragionare per astrazioni
- Il C consente, in particolare, di:
  - ridefinire tipi già esistenti
  - definire dei nuovi *tipi enumerativi*
  - definire dei nuovi *tipi strutturati*

TIPI RIDEFINITI

- Un nuovo identificatore di tipo viene dichiarato identico a un tipo già esistente
- Schema generale:

```
typedef  TipoEsistente NuovoTipo ;
```

- Esempio

```
typedef     int MioIntero;
MioIntero   X,Y,Z;
int         W;
```

TIPI ENUMERATIVI

- Un tipo enumerativo viene specificato tramite *l'elenco dei valori* che i dati di quel tipo possono assumere.
- Schema generale:

```
typedef enum {
   a1, a2, a3, ... , aN } EnumType;
```

- Il compilatore associa a ciascun "identificativo di valore" a1,..aN un *numero naturale* (0,1,...), che viene usato nella valutazione di espressioni che coinvolgono il nuovo tipo.

TIPI ENUMERATIVI

- Gli "identificativi di valore" `a1,... , aN` sono a tutti gli effetti *delle nuove costanti*.
- Esempi

```
typedef enum {
  lu, ma, me, gi, ve, sa, do} Giorni;
typedef enum {
  cuori, picche, quadri, fiori} Carte;
Carte   C1, C2, C3, C4, C5;
Giorni  Giorno;
if (Giorno == do) /* giorno festivo */
else /* giorno feriale */
```

TIPI ENUMERATIVI

- **Un "identificativo di valore" può comparire *una sola volta* nella definizione di *un solo tipo*, altrimenti si ha ambiguità.**
- **Esempio**

```
typedef enum {
  lu, ma, me, gi, ve, sa, do} Giorni;
typedef enum { lu, ma, me} PrimiGiorni;
```

La definizione del secondo tipo enumerativo è ***scorretta***, perché gli identificatori `lu, ma, me` sono già stati usati altrove.

TIPI ENUMERATIVI

- **Un tipo enumerativo è *totalmente ordinato:* vale l'ordine con cui gli identificativi di valore sono stati elencati nella definizione.**
- **Esempio**

```
typedef enum {
  lu, ma, me, gi, ve, sa, do} Giorni;
```

Data la definizione sopra,

`lu < ma` *è vera*

`lu >= sa` *è falsa*

in quanto `lu` ↔ 0, `ma` ↔ 1, `me` ↔ 2, etc.

TIPI ENUMERATIVI

- **Poiché un tipo enumerativo è, *per la macchina C,* indistinguibile da un intero, è possibile in linea di principio *mischiare interi e tipi enumerativi***
- **Esempio**

```
typedef enum {
  lu, ma, me, gi, ve, sa, do} Giorni;
Giorni g;
g = 5;        /* equivale a g = sa */
```

- ***È una pratica da evitare ovunque possibile!***

TIPI ENUMERATIVI

- **È anche possibile specificare i valori naturali cui associare i simboli `a1,...,aN`**
- qui, `lu` ↔ 0, `ma` ↔ 1, `me` ↔ 2, etc.:

```
typedef enum {
  lu, ma, me, gi, ve, sa, do} Giorni;
```

- qui, invece, `lu` ↔ 1, `ma` ↔ 2, `me` ↔ 3, etc.:

```
typedef enum {
  lu=1, ma, me, gi, ve, sa, do} Giorni;
```

- qui, infine, l'associazione è data caso per caso:

```
typedef enum { lu=1, ma, me=7, gi, ve,
  sa, do} Giorni;
```

IL TIPO BOOLEAN

- **Il boolean non esiste in C, ma si può facilmente definirlo:**

```
typedef enum { false, true } Boolean;
```

- così:

  `false` ↔ 0, `true` ↔ 1

  `false` < `true`

- logica positiva

EQUIVALENZA

- **La possibilità di introdurre nuovi tipi pone il problema di stabilire *se e quanto* due tipi siano *compatibili fra loro.***
- **Due possibili scelte:**
  - **equivalenza *strutturale***

    tipi equivalenti se *strutturalmente identici*
  - **equivalenza *nominale***

    tipi equivalenti se *definiti nella stessa definizione* oppure se *il nome dell'uno è definito espressamente come identico all'altro.*

Scelta dal C

EQUIVALENZA STRUTTURALE

- **Esempio di equivalenza *strutturale***

```
typedef int  MioIntero;
typedef int  NuovoIntero;
MioIntero    A;
NuovoIntero  B;
```

- **I due tipi `MioIntero` e `NuovoIntero` sono equivalenti perché *strutturalmente identici* (sono entrambi `int` per la macchina C)**
- Quindi, A=B è un assegnamento lecito.

EQUIVALENZA NOMINALE

- **Non è il caso del C, ma è il caso, per esempio, del Pascal**
- **Esempio di equivalenza *nominale***

```
type MioIntero = integer;
type NuovoIntero = integer;
var A: MioIntero;
var B: NuovoIntero;
```

- **I due tipi `MioIntero` e `NuovoIntero` *non sono equivalenti* perché *definiti in una diversa definizione* (`A:=B` *è rigettata!)***

DEFINIZIONE DI TIPI STRUTTURATI

- **Abbiamo visto a suo tempo come introdurre *variabili* di tipo array e struttura:**

```
char msg1[20], msg2[20];
struct persona {...} p,q;
```

- **Non potendo però *dare un nome* al nuovo tipo, dovevamo *ripetere la definizione* per ogni nuova variabile**
  - per le strutture potevamo evitare di ripetere la parte fra {..}, ma `struct persona` andava ripetuto comunque.

DEFINIZIONE DI TIPI STRUTTURATI

- **Ora possiamo *definire un nuovo tipo* array e struttura, come segue:**

```
typedef char string[20];
typedef struct {... } persona;
```

- **ciò consente, d'ora in poi, di *non dover più ripetere la definizione per esteso* ogni volta che si definisce una nuova variabile:**

```
string s1, s2;  /* due stringhe di 20 caratteri */
persona p1, p2;  /* due strutture "persona" */
```

  - per le strutture, ciò rende quasi sempre inutile specificare una etichetta dopo la parola chiave `struct`

DEFINIZIONE DI NUOVI TIPI

Perché definire nuovi tipi è importante?

- **Permette di progettare *al nostro livello di astrazione*, *non a quello della macchina C***
  - Il mondo reale è fatto di *giorni, temperature, colori, stringhe...* **non di int, char, etc!!**
  - operare su "stringhe", "matrici", "vettori" è ben diverso che operare su char x[20], etc
- **Consente di *prescindere dalla rappresentazione concreta delle cose***
  - temperature, colori, interi... saranno anche tutti int, alla fine... **ma concettualmente sono diversi!**

TIPI DI DATO ASTRATTO

Un *tipo di dato astratto (ADT)* definisce una categoria concettuale con le sue proprietà:

- una ***definizione di tipo***
  - **implica un dominio, D**
- un ***insieme di operazioni ammissibili* su oggetti di quel tipo**
  - **funzioni: *calcolano valori sul dominio D***
  - **predicati: *calcolano proprietà vere o false su D***

TIPI DI DATO ASTRATTO IN C

In C, un *ADT* si costruisce definendo:

- ***il nuovo tipo con*** `typedef`
- ***una funzione per ogni operazione***

Esempio: il contatore

- una entità caratterizzata da un valore intero

```
typedef int counter;
```

- con operazioni per
  - resettare il contatore a zero `reset(counter*);`
  - incrementare il contatore `inc(counter*);`

ORGANIZZAZIONE DI ADT IN C

La struttura di un ADT comprende quindi:

- ***un file header, contenente***
  - ***la*** `typedef`
  - ***la dichiarazione delle funzioni***
- ***un file di implementazione, contenente***
  - ***una direttiva #include per includere il proprio header*** (per importare la definizione di tipo)
  - ***la definizione delle funzioni***

L'ADT `counter`

- ***file header*** `counter.h`

```
typedef int counter;
void reset(counter*);
void inc(counter*);
```

Definisce in astratto cos'è un counter e cosa si può fare con esso.

- ***file di implementazione*** `counter.c`

```
#include "counter.h"
void reset(counter c*){ *c=0; }
void inc(counter*){ (*c)++; }
```

Ha bisogno di sapere cos'è un counter, e specifica *come funziona.*

L'ADT `counter`: un cliente

Per usare un `counter` occorre

- ***includere il relativo file header***
- ***definire una o più variabili di tipo*** `counter`
- ***operare su tali "oggetti" mediante le sole operazioni (funzioni) previste***

```
#include "counter.h"
main(){
  counter c1, c2;
  reset(&c1); reset(&c2);
  inc(&c1); inc(&c2); inc(&c2);
}
```

OPERAZIONI DI UN ADT

Quali operazioni definire per un ADT?

- **costruttori**
  - **costruiscono un oggetto di questo tipo (a partire dai suoi "costituenti elementari")**
- **selettori**
  - **restituiscono uno dei "mattoni elementari" che compongono l'oggetto**
- **predicati**
  - **verificano la presenza di una proprietà sull'oggetto, restituendo *vero* o *falso***

OPERAZIONI DI UN ADT

Quali operazioni definire per un ADT?

- **funzioni**
  - **agiscono in vario modo sugli oggetti**
- ***trasformatori***
  - ***cambiano lo stato dell'oggetto***
  - **Possono esserci o no: se non ci sono, l'oggetto *non ha stato,* ossia *non è modificabile.***
  - ***Decidere se fare un oggetto con o senza stato è una scelta cruciale di progetto: mai farla a caso!***

ESERCIZIO

Realizzare l'ADT che cattura il concetto di "stringa di al più 250 caratteri"

- **realizzazione fisica basata su un array di caratteri (ed eventualmente la dimensione)**
- **definire le operazioni per**
  - **estrarre il carattere situato alla i-esima posizione: *SELETTORE***
  - **calcolare la lunghezza *FUNZIONE***
  - **creare una nuova stringa che sia la concatenazione di due stringhe date *COSTRUTTORE***
  - **confrontare due stringhe *FUNZIONE***

ESERCIZIO

Realizzare l'ADT che cattura il concetto di "insieme" (di interi)

- **realizzazione fisica basata, ad esempio, su una struttura con un array e un indice**
- **operazioni per**
  - **aggiungere un elemento, togliere un elemento *TRASFORMATORI***
  - **verificare la presenza di un elemento *PREDICATO***
  - **calcolare l'unione di due insiemi, la differenza fra due insiemi, l'intersezione, etc**
- **ricorda:** un insieme non ha una nozione di ordine, e non può avere elementi duplicati

ADT IN C: LIMITI

- **Gli ADT così realizzati funzionano, ma *molto dipende dall'autodisciplina del programmatore***
- **Non esiste alcuna protezione contro un uso scorretto dell'ADT**
  - l'organizzazione *suggerisce* di operare sull'oggetto solo tramite le funzioni previste, ma *non riesce a impedire* di aggirarle a chi lo volesse
- **La struttura interna dell'oggetto *è sotto gli occhi di tutti*** (nella `typedef`)

ADT IN C: LIMITI

Superare questi limiti sarà uno degli obiettivi cruciali della *programmazione a oggetti*, che vedrete nel corso di Fondamenti B con il linguaggio Java.