

L'astrazione nella programmazione

Principio di astrazione...

Riepilogando, negli anni '50 e '60 si introdussero diverse forme di astrazione nei linguaggi di programmazione (strutture di controllo, operatori, dati).

Si giunse quindi a maturare la convinzione che:

“È possibile costruire astrazioni su una qualunque classe sintattica, purché le frasi di quella classe specifichino un qualche tipo di computazione”.

...Principio di astrazione

Di seguito si esaminerà l'applicazione del principio di astrazione a sei classi sintattiche, in particolare:

- Espressione → astrazione di funzione
- Comando → astrazione di procedura
- Controllo di sequenza → astrazione di controllo
- Accesso a un'area di memoria → astrazione di selettore
- Definizione di un dato → astrazione di tipo
- Dichiarazione → astrazione generica

Tutte queste classi sintattiche sottintendono una computazione ...

...Principio di astrazione.

Infatti, ...

- L'astrazione di una *funzione* include una espressione da valutare.
- L'astrazione di una *procedura* include un comando da eseguire.
- L'astrazione di *controllo* include delle espressioni di controllo dell'ordine di esecuzione delle istruzioni.
- L'astrazione di *selettore* include il calcolo dell'accesso ad una variabile
- L'astrazione di *tipo* include un gruppo di operatori che definiscono implicitamente un insieme di valori.
- L'astrazione *generica* include una frase che sarà elaborata per produrre legami (binding).

Controesempio:

La classe sintattica “letterale” non può essere astratta perché non specifica alcun tipo di computazione.

Astrazione di funzione

Un'astrazione di funzioni include un'**espressione** da valutare, e quando chiamata, darà un **valore** come risultato.

Un'astrazione di funzione è specificata mediante una definizione di funzione, del tipo:

function I(FP₁; ...; FP_n) is E

dove **I** è un identificatore, **FP₁; ...; FP_n** sono parametri formali, ed **E** è l'espressione da valutare.

In questo modo si lega **I** a un'entità, l'astrazione di funzione, che ha la proprietà di dare un risultato ogni qualvolta è chiamata con argomenti appropriati.

Astrazione di funzione

Una chiamata di funzione, $I(AP_1; \dots; AP_n)$ dove i parametri effettivi, $AP_1; \dots; AP_n$ determinano gli argomenti, ha due punti di vista:

- Punto di vista dell'*utente*: la chiamata di una funzione trasforma gli argomenti in un risultato;
- Punto di vista dell'*implementatore*: la chiamata valuta E , avendo precedentemente vincolato i parametri formali agli argomenti corrispondenti. L'algoritmo codificato in E è di interesse per l'implementatore.

Anomalie di progetto dei linguaggi

Se l'astrazione di funzione include un'espressione **E** da valutare, è naturale attendersi che il corpo della funzione non includa dei comandi (come le assegnazioni, le istruzioni di salto, le iterazioni, ecc.) il cui effetto è quello di cambiare lo stato di un sistema, non di produrre valori.

Non sempre ciò accade, come si può osservare in questo esempio di funzione scritta in Pascal.

```
function power(x:real;n:integer) : real
begin
  if n= 1 then power := x
  else power := x*power(x,n-1)
end
```

Anomalie di progetto dei linguaggi

- Nel corpo della funzione Pascal compaiono i **comandi di assegnazione**.
- Non è proprio possibile farne a meno, perché per poter restituire un valore, in Pascal è necessario assegnare un valore a una pseudo variabile, che ha lo stesso nome della funzione.
- Nell'esempio: l'identificatore della funzione, **power**, può denotare sia l'espressione da valutare che la pseudo variabile dove sarà depositato il risultato.
- Il corpo della funzione è sintatticamente un comando, ma semanticamente è una espressione (la funzione, infatti può essere invocata solo alla destra di operazioni di assegnazione).

Anomalie di progetto dei linguaggi

- Comandi nel corpo di funzioni. Necessariamente così?
- No. Riportiamo di seguito un esempio nel linguaggio funzionale ML.

```
function power(x:real;n:int) is  
  if n= 1  then x  
  else x*power(x,n-1)
```

Nel corpo della funzione definita in ML compare una espressione condizionale la cui valutazione non modifica lo stato del sistema.

Anomalie di progetto dei linguaggi

- Ma allora perché Pascal, Ada e altri linguaggi di programmazione permettono di avere comandi nel corpo di funzioni?
- Per poter sfruttare la potenza espressiva dell'assegnazione e dell'**iterazione** nella computazione di risultati.
- Diversamente saremmo costretti a esprimere ricorsivamente le espressioni da valutare. E la ricorsione, com'è noto, può essere fonte di inefficienze nell'uso delle risorse di calcolo.

Anomalie di progetto dei linguaggi

Morale:

Molti linguaggi permettono di avere comandi nel corpo di funzioni. Questo per ragioni di efficienza.

Si lascia al programmatore il compito di utilizzare correttamente i comandi in modo da evitare che la funzione possa avere effetti collaterali (*side effect*) oltre quello di calcolare un valore.



Riflessioni

La funzione *getchar(f)*, che legge un carattere dal file *f* e restituisce quel carattere ha un effetto collaterale su *f*.

Cosa fa il seguente codice, supposto che *gender* sia una variabile enumerativa che assume valori *female*, *male* e *undefined*?

```
if getchar(f) = 'F' then
```

```
    gender := female
```

```
else if getchar(f)='M' then
```

```
    gender := male
```

```
else gender := undefined.
```



Astrazione di funzione

Le funzioni, in quanto astrazioni di espressioni, possono comparire ovunque occorra un'espressione.

Pertanto, esse compaiono alla destra di operazioni di assegnazioni, ma anche al posto di parametri effettivi nelle chiamate di altre funzioni (o procedure), laddove un valore potrebbe essere calcolato mediante una espressione.

$$y := f(y, \text{power}(x, 2))$$

La funzione f ha due parametri passati per valore.

Astrazione di funzione

In molti linguaggi di programmazione è possibile definire dei parametri formali che sono un riferimento a una funzione.

Esempio: in Pascal

```
function Sommatoria(function F(R: real, M: integer): real; X: real;  
    N:integer): real;  
var I: integer;  
    Sum: real;  
begin  
    Sum := 0;  
    for I := 1 to N do Sum := Sum + F(x, I);  
    Sommatoria := Sum  
end
```

Astrazione di funzione

Esempio (cont.)

Y := Sommatoria(**power**, x, n);

calcola la sommatoria: $\sum_1^n x^i$

In alcuni linguaggi di programmazione si separa il concetto di astrazione di funzione da quello di legame (binding) a un identificatore. Questo permette di passare come parametri delle funzioni anonime (prive di identificatore).

Infine, le funzioni possono essere il **risultato** di valutazioni di espressioni o possono essere assegnate a variabili:

val cube = fn (x: real) => x*x*x ← in ML

Astrazione di funzione

Riepilogando:

In alcuni linguaggi di programmazione (Fortran, Ada-83) le funzioni sono di **terza classe**, cioè possono essere solo chiamate, in altri (vedi Pascal) sono di **seconda classe**, cioè possono essere passate come argomenti, mentre in altri linguaggi di programmazione esse sono di **prima classe** in quanto possono essere anche restituite come risultato della chiamata di altre funzioni o possono essere assegnate come valore a una variabile (alcuni linguaggi di programmazione funzionali, come Lisp e ML, e linguaggi di scripting, come Perl, permettono di generare funzioni al run-time).

Cittadini di prima classe

In generale, in programmazione una qualunque entità (dato, procedura, funzione, etc.) si dice **cittadino di prima classe** quando non è soggetta a restrizioni nel suo utilizzo.



Cittadini di prima classe

A tal proposito, osserviamo che i valori possono essere:

- *Denotabili*, se possono essere associati ad un nome;
- *Esprimibili*, se possono essere il risultato di un'espressione complessa (cioè diversa da un semplice nome)
- *Memorizzabili*, se possono essere memorizzati in una variabile.

Cittadini di prima classe

Esempio:

Nei linguaggi imperativi, i valori di tipo intero sono in genere sia denotabili, che esprimibili che memorizzabili.

Al contrario, i valori del tipo delle funzioni da Integer a Integer sono denotabili in quasi tutti i linguaggi, perché possiamo dare loro un nome con una dichiarazione:

```
int succ(int x) { return x+1 }
```

ma **non sono esprimibili** in quasi tutti i linguaggi imperativi, perché non ci sono espressioni complesse che restituiscono una astrazione di funzione come risultato della loro valutazione. Allo stesso modo **non sono valori memorizzabili**, perché non possiamo assegnare una funzione ad una variabile.

Cittadini di prima classe

Esempio (cont.)

La situazione è diversa per i linguaggi di altri paradigmi, quali ad esempi i linguaggi funzionali (Scheme, ML, haskell, ecc.), nei quali i valori funzionali sono sia denotabili, che esprimibili, che, in certi linguaggi, memorizzabili.

Stessa cosa dicasi per gli **oggetti** ai quali si accennerà in seguito. Nei linguaggi imperativi essi sono denotabili ma non esprimibili e memorizzabili. In altri termini, gli oggetti non sono cittadini di prima classe. Al contrario, nei linguaggi orientati agli oggetti, gli oggetti sono denotabili, esprimibili e memorizzabili, cioè sono cittadini di prima classe.

Astrazione di procedura

Un'astrazione di procedura include un **comando** da eseguire, e quando chiamata, aggiornerà le variabili che rappresentano lo stato del sistema.

Un'astrazione di procedura è specificata mediante una definizione di procedura, del tipo:

procedure $I(FP_1; \dots; FP_n)$ is C

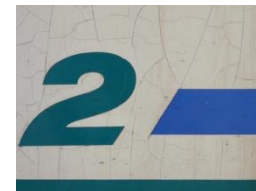
dove **I** è un identificatore, **$FP_1; \dots; FP_n$** sono parametri formali, e **C** è il blocco di comandi da eseguire.

In questo modo si lega **I** all'astrazione di procedura, che gode della proprietà di cambiare lo stato del sistema quando chiamata con argomenti appropriati.

Astrazione di procedura

Data una chiamata di procedura, $I(AP_1; \dots; AP_n)$ dove $AP_1; \dots; AP_n$ sono i parametri effettivi, il punto di vista dell'*utente* è che la chiamata aggiornerà lo stato del sistema in modo dipendente dai parametri, mentre il punto di vista dell'*implementatore* è che la chiamata consentirà l'esecuzione del corpo di procedura **C**, avendo precedentemente vincolato i parametri formali agli argomenti corrispondenti. L'algoritmo codificato in **C** è di interesse solo per l'implementatore.

In Pascal e in C le procedure sono *cittadini di seconda classe*.



Astrazione di procedura

Esempio di puntatori a funzioni in C

La funzione **bubble** ordina un array di interi sulla base di una funzione di ordinamento.

L'argomento con puntatore a funzione

```
void (*compare) ( int, int, int *)
```

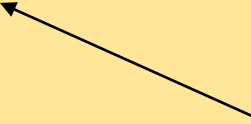
Dice a **bubble** di aspettarsi un puntatore a una funzione, identificata da **compare**, che prende tre argomenti in ingresso e restituisce un tipo **void**.

N.B.: Se avessimo rimosso le parentesi

```
void *compare( int, int, int * )
```

avremmo dichiarato semplicemente una funzione che prende in input tre interi e restituisce un puntatore a **void**.

```
1/* Fig. 7.26: fig07 26.c
2 Multipurpose sorting program using function pointers */
3#include <stdio.h>
4#define SIZE 10
5void bubble( int [], const int, void (*)( int, int, int * ) );
6void ascending( int, int, int* );
7void descending( int, int, int * );
8
9int main()
10 {
11
12     int order,
13         counter,
14         a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16     printf( "Enter 1 to sort in ascending order,\n"
17            "Enter 2 to sort in descending order: " );
18     scanf( "%d", &order );
19     printf( "\nData items in original order\n" );
20
21     for ( counter = 0; counter < SIZE; counter++ )
22         printf( "%5d", a[ counter ] );
23
24     if ( order == 1 ) {
25         bubble( a, SIZE, ascending );
26         printf( "\nData items in ascending order\n" );
27     }
28     else {
29         bubble( a, SIZE, descending );
30         printf( "\nData items in descending order\n" );
31     }
32 }
```



Si noti il parametro di tipo puntatore a funzione.


```

33  for ( counter = 0; counter < SIZE; counter++ )
34      printf( "%5d", a[ counter ] );
35
36  printf( "\n" );
37
38  return 0;
39 }
40
41 void bubble( int work[], const int size,
42             void (*compare)( int, int, int * ) )
43 {
44     int pass, count, bool;
45
46     void swap( int *, int * );
47
48     for ( pass = 1; pass < size; pass++ )
49
50         for ( count = 0; count < size - 1; count++ )
51             (*compare)( work[ count ], work[ count + 1 ], &bool);
52             if ( bool )
53                 swap( &work[ count ], &work[ count + 1 ] );
54 }
55
56 void swap( int *element1Ptr, int *element2Ptr )
57 {
58     int temp;
59
60     temp = *element1Ptr;
61     *element1Ptr = *element2Ptr;
62     *element2Ptr = temp;
63 }
64

```

ascending e descending restituiscono in **bool** il risultato che può essere 0 o 1. **bubble** chiama **swap** se **bool** è 1.

Nota come i puntatori a funzione sono chiamati usando l'operatore di dereferenziazione (*), che seppur non richiesto, enfatizza come **compare** sia un puntatore a funzione e non una funzione.

```
65 void ascending( int a, int b, int * c )
66 {
67     *c = b < a;    /* swap if b is less than a */
68     return
69 }
70 void descending( int a, int b, int * c )
71 {
72     *c = b > a;    /* swap if b is greater than a */
73     return }
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1
```

Data items in original order

2 6 4 8 10 12 89 68 45 37

Data items in ascending order

2 4 6 8 10 12 37 45 68 89

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2
```

Data items in original order

2 6 4 8 10 12 89 68 45 37

Data items in descending order

89 68 45 37 12 10 8 6 4 2

Astrazione funzionale

L'astrazione di funzione o di procedura sono *due tecniche di programmazione* di supporto all'**astrazione funzionale**, intesa come *tecnica di progettazione del software* secondo la quale occorre distinguere la specifica di un operatore (come esso è visto e manipolato dall'utente) dalla sua realizzazione.

Quale sia la tecnica di programmazione (astrazione di funzione o astrazione di procedura) più opportuna da adottare dipende da:

- **Tipo di operatore progettato:**
 - ha effetti collaterali → astrazione di procedura
 - non ha effetti collaterali → astrazione di funzione

Astrazione funzionale

- Limiti imposti dal linguaggio di programmazione.

Esempio: se l'operatore *bubblesort* non modifica l'array passato in ingresso, ma ne restituisce uno ordinato, sarebbe più opportuno implementarlo ricorrendo a un'astrazione di funzione, in quanto restituisce un nuovo valore che è un intero array ordinato. Tuttavia, non tutti i linguaggi di programmazione permettono di definire funzioni che restituiscono dati qualsivoglia complessi. In alcuni linguaggi (vedi Pascal) gli array sono cittadini di seconda classe. In tal caso l'operatore *bubblesort* dovrà essere necessariamente realizzato mediante astrazione di procedura.

Astrazione di controllo

L'astrazione di controllo si applica alla classe sintattica struttura di controllo.

Le strutture di controllo definiscono l'ordine in cui le singole istruzioni o i gruppi di istruzioni (unità di programma) devono essere eseguiti.

Il linguaggio macchina generalmente fornisce due semplici meccanismi per governare il flusso di controllo delle istruzioni singole: **l'elaborazione in sequenza** e il **salto**.

Nei linguaggi assemblativi le istruzioni da eseguire in sequenza sono scritte l'una dopo l'altra (sottointendendo che vanno scritte in locazioni di memoria contigue); il salto è rappresentato da un'istruzione di jump:

jump to <indirizzo simbolico o label>

Astrazione di controllo

L'indirizzo simbolico è comunque un dettaglio di scarsa importanza per il programmatore: quello che conta è poter indicare le prossime istruzioni da eseguire.

Per questa ragione i linguaggi di alto livello hanno introdotto strutture di controllo astratte come la **selezione**:

if cond **then** S1
 else S2

jump on <cond> to A

S2

jump to B

A: S1

B: ...

Astrazione di controllo

Similmente sono state introdotte diverse strutture di controllo astratte per l'**iterazione**.

Inoltre l'utilizzo dello stack per conservare gli indirizzi di ritorno dalle chiamate di funzione/procedura si è tradotta nella possibilità di effettuare chiamate **ricorsive**.

Astrazione di controllo

Di particolare interesse è l'attuale tendenza a offrire strutture di controllo iterative per quei dati astratti che sono collezioni omogenee (e.g., insiemi, multiinsiemi, liste, array) di valori.

La struttura di controllo iterativa ha due parametri, il dato astratto e la variabile alla quale assegnare un valore contenuto nel dato astratto. Il meccanismo di visita della collezione di valori è incapsulato nel dato astratto, che dispone di operazioni lecite per consentire l'iterazione sulla collezione di oggetti.

Astrazione di controllo

Esempio: In Java è disponibile l'astrazione di controllo **for-each**, che permette di iterare su una collezione.

```
LinkedList<Integer> list = new LinkedList<Integer>();  
... /* si inseriscono degli elementi  
for (Integer n : list) {  
    System.out.println(n);  
}
```

L'operazione messa a disposizione di **LinkedList** per l'astrazione di controllo **for-each** è **iterator()**.

Astrazione di controllo

Se guardiamo alle strutture di controllo come espressioni che, quando valutate, definiscono l'ordine in cui eseguire dei comandi, possiamo specificare un'astrazione di controllo come segue:

control I(FP₁; ...; FP_n) is S

dove **I** è un identificatore di una nuova struttura di controllo, **FP₁; ...; FP_n** sono parametri formali, e **S** è una espressione di controllo che definisce un ordine di esecuzione.

Astrazione di controllo

Esempio: (in un ipotetico linguaggio di programmazione)

control swap(boolean: cond, statement: S1,S2) is

if *cond* then

begin *S1;S2* end

else

S2;S1

endif

Argomento di tipo
statement

Argomento di tipo
espressione

La chiamata `swap(i<j, {i:=j}, {j:=i})` porta i al valore di j se i è minore di j, altrimenti porta j al valore di i.

Astrazione di controllo

Esempio: (in un ipotetico linguaggio di programmazione)

control alt_execution(statement: S1,S2,S3) is

if abnormal (*S1*) then

S2

else

S3

endif



Argomento di tipo
statement

abnormal è un predicato che ha in input un comando S1 e restituisce *true* se l'esecuzione di S1 è terminata in modo anomalo (è stata sollevata un'eccezione), *false* altrimenti.

Astrazione di controllo

I linguaggi di programmazione moderni mettono a disposizione dei meccanismi sofisticati per gestire le situazioni eccezionali come:

- Errori aritmetici (e.g., divisione per 0) o di I/O (e.g., leggi un intero ma ottieni un carattere)
- Fallimento di precondizioni (e.g., prelievo da una coda vuota),
- Condizioni imprevedibili (e.g., lettura di un fine file)

Quando viene **sollevata** un'**eccezione**, essa dev'essere **catturata** e **gestita**.

Astrazione di controllo

Tipici approcci per **localizzare il gestore dell'eccezione** sono:

- Cerca un gestore nel blocco (o sottoprogramma) corrente
- Se non c'è, forza l'uscita dall'unità corrente e solleva l'eccezione nell'unità chiamante
- Continua a risalire la gerarchia delle unità chiamanti finché non trovi un gestore oppure non raggiungi il livello più alto (main).

Cosa accade quando il gestore è trovato e l'eccezione è trattata?

- Si riparte dall'unità contenente il gestore (**modello di terminazione**), come in Ada.
- Si ritorna ad eseguire il comando che ha generato l'eccezione (**modello di ripresa**, *resumption*), come in PL/I

Astrazione di controllo

- I linguaggi di programmazione sequenziale utilizzano la sequenza, la selezione e la ripetizione per definire un **ordinamento totale** sulla esecuzione dei comandi in un programma.
- I linguaggi di programmazione paralleli utilizzano ulteriori costrutti del flusso di controllo, come ***fork***, ***cobegin***, o cicli ***for*** paralleli, per introdurre in modo esplicito un **ordinamento parziale** sulla esecuzione dei comandi.
- Dato che il parallelismo è una forma di controllo della sequenza di esecuzione, l'astrazione di controllo è particolarmente importante nella programmazione parallela.

Astrazione di selettore

* IN MEMORIA

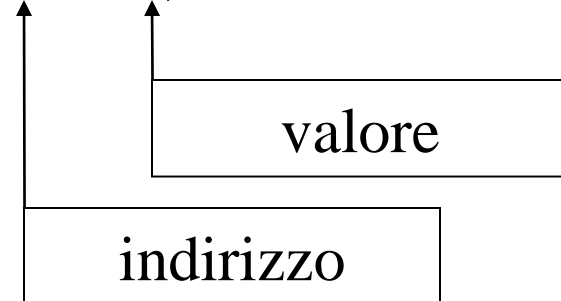
- Un linguaggio di programmazione dispone di costrutti per poter accedere^{*} ad una variabile (strutturata e non). Ad esempio in Pascal abbiamo:

ACCESSO PER VALORE O PER INDIRIZZO

var r: real;

selettore
...

r := r*r;



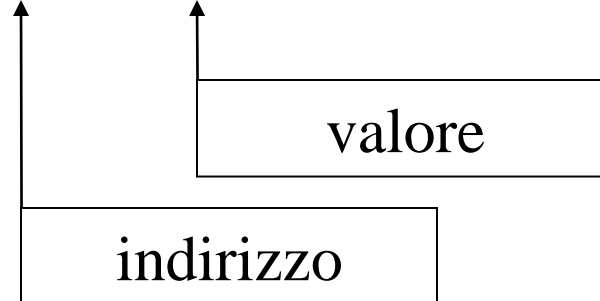
var vett: array[1..10] of char;

...

vett[i] := vett[i+1];

Due selettori: vett e []

Con vett accedo all'indirizzo del vettore, con le quadre accedo alle celle successive in base all'indice.



Selettore variabile utilizzato in espressioni e comandi. Per esempio in un assegnamento, quando è a sinistra è veramente un selettore perchè deve accedere ad un'area di memoria. Quando è a destra invece prende un valore da memorizzare.

Astrazione di selettore

- Lo stesso identificatore può essere usato sia come nome di un valore (legame dinamico fra identificatore e valore) e sia come un indirizzo (legame statico fra identificatore e locazione di memoria). Identicamente possiamo avere:

```
type rec= record
```

```
  a: real;
```

Campi di record sono selettori

```
  b:real
```

```
end
```

```
var r: rec;
```

```
...
```

```
r.b := r.a*3;
```

Un accesso a una variabile restituisce un *riferimento* a una variabile.

Astrazione di selettore

- Tuttavia, se osserviamo il seguente codice Pascal:

```
type queue= ...;
```

```
var Aq : queue;
```

```
function first(q:queue): integer
```

```
... (* restituisce il primo intero  
della coda *)
```

```
...
```

```
i := first(Aq);
```

ci accorgiamo che la chiamata `first (Aq)` può comparire solo alla destra di una assegnazione, perché le funzioni restituiscono valori, mentre a sinistra:

▷ `first (Aq) := 0;`

NO IN PASCAL

dovrebbe restituire riferimenti ad aree di memoria.

Astrazione di selettore

In Pascal abbiamo dei selettori predefiniti dal progettista del linguaggio:

- F^{\wedge} : riferimento a un puntatore F
- $V[E]$: riferimento a un elemento di un array V
- $R.A$: riferimento a un elemento di un record

ma il programmatore non ha modo di definire un nuovo selettore, come il riferimento a una lista di elementi indipendentemente da come la lista è realizzata.

In altri termini, non c'è la possibilità di definire astrazioni di selettore, che restituiscano l'accesso a un'area di memoria.

Astrazione di selettore

Per poter scrivere una assegnazione del tipo:

`first (Aq) := 0;`

dobbiamo poter definire un tipo di astrazione che quando chiamata restituisce il riferimento a una variabile (astrazione di selettore).

Supponiamo di estendere il Pascal con le astrazioni di selettore:

selector $I(FP_1; \dots; FP_n)$ is A

dove **A** è una espressione che restituisce un accesso a una variabile (che denoteremo con **$\&$** come in C).

Astrazione di selettore

Potremo allora definire **first** come segue:

```
type queue= record
  elementi: array[1..max] of integer;
  testa, fondo, lung: 0..max;
end;
selector first(q:queue) is &(q.elementi[q.testa]);
```

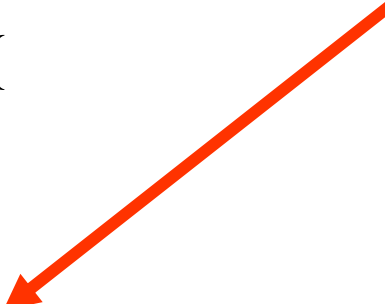
Questo ci consentirebbe di scrivere espressioni come:

```
first (Aq) :=first (Aq) +1;
```

dove l'invocazione di destra si riferisce alla funzione mentre quella di sinistra si riferisce all'astrazione di selettore.

Astrazione di selettore in C++

```
class Queue {  
...  
public:  
    static int & first(Queue *);  
};  
Queue q*;  
...  
int i=first(q);  
first(q) *= first(q);  
first(q)++;
```



OBBLIGO PER ESSERE UN SELETTOR

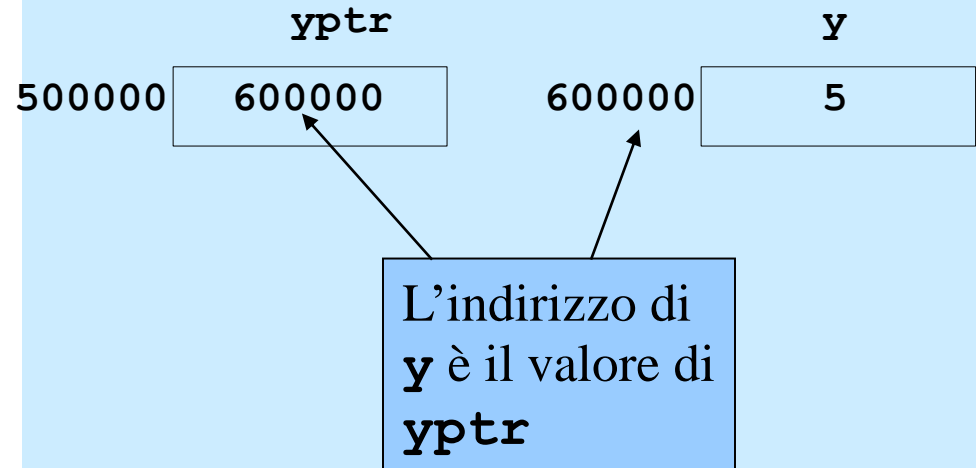
& (operatore di indirizzamento)

– Restituisce l'indirizzo di un operando

```
int y = 5;
```

```
int *yPtr;
```

```
yPtr = &y;
```



Astrazione di selettore in C++

Una possibile implementazione:

```
class Queue {  
int items[100];  
int front, rear;  
public:  
...  
int & first(Queue *q){ return q->items[front]; }  
};
```

I dettagli dell'implementazione non sono di interesse per l'utente. Queue avrebbe potuto essere implementata diversamente.

Astrazione di selettore in C++

Un altro esempio: Selettore su una lista concatenata

```
int & get( List *p, int el ) { /* lista concatenata */  
    int i ;  
    for ( i = 1; i < el ; i ++ ) {  
        p = p-> next ; /* prende l'elemento successivo */  
    }  
    return p-> data ;  
}
```

...

```
get(head, i ) = get(head, 2) + 1;
```

...

Flessibilità dei linguaggi

L'espressività di un linguaggio di programmazione dipende direttamente da:

- I *meccanismi di composizione*, ossia la definizione di operazioni complesse a partire da quelle semplici.
- I *meccanismi di controllo di sequenza*, ossia la possibilità di stabilire ordini di esecuzione in modo semplice.
- I *valori* che il linguaggio permette di rappresentare e manipolare come dati.

I linguaggi che supportano ...

- l'astrazione funzionale sono flessibili nei meccanismi di composizione;
- l'astrazione di controllo sono flessibili nei meccanismi di controllo di sequenza;
- l'*astrazione dati* sono flessibili nella definizione e manipolazione di nuovi valori.

Tecniche di programmazione a supporto dell'astrazione dati

Le tecniche di **programmazione** a supporto dell'astrazione dati, che è una tecnica di progettazione del software, sono due:

- Definizione di **tipi astratti**, cioè l'astrazione sulla classe sintattica tipo. ↳ ASTRAZIONE DI TIPO
- Definizione di **classi di oggetti**, cioè l'astrazione sulla dichiarazione di moduli dotati di stato locale. ↳ ASTRAZIONE BENEFICIA

SIAMO SEMPRE NEL PARADIGMA IMPERATIVO

Tecniche di programmazione a supporto dell'astrazione dati

In entrambi i casi occorre poter **incapsulare** la rappresentazione del dato con le operazioni lecite. Tuttavia,

Tipo astratto → il modulo rende visibile all'utilizzatore sia un identificatore di tipo che degli operatori.

Classe di oggetti → il modulo rende visibile all'utilizzatore solo gli operatori.

Inoltre:

Tipo astratto T → i valori sono associati a variabili dichiarate di tipo T.

Classe di oggetti C → i valori sono associati a oggetti ottenuti per istanziazione della classe oggetti C.

Tipo concreto e tipo astratto

I linguaggi ad alto livello mettono a disposizione del programmatore un nutrito gruppo di tipi predefiniti, detti **concreti**. Essi si distinguono in *primitivi* o *semplici* (cioè, i valori associati al tipo sono atomici) e *composti* o *strutturati* (i cui valori sono ottenuti per composizione di valori più semplici).

Tuttavia un linguaggio di programmazione sarà tanto più espressivo quanto più semplice sarà per il programmatore definire dei *suoi* tipi di dato a partire dai tipi di dato concreti disponibili. I tipi definiti dall'utente (detti anche *user defined types*, UDT) sono anche detti **astratti**.

Tipo concreto e tipo astratto



Astrazione di tipo

L'**espressione di tipo** (spesso abbreviato con **tipo**) è il costrutto con cui alcuni linguaggi di programmazione consentono di definire un nuovo tipo.

Esempio: in Pascal

type Person = record

 name: **packed array**[1..20] **of** char;

 age: integer;

 height: real

end

In questo caso si stabilisce **esplicitamente** una **rappresentazione** per i valori del tipo *Person* e **implicitamente** gli **operatori** applicabili a valori di quel tipo.

Per forza di cose, gli operatori del tipo *Person* dovranno essere generici (come l'assegnazione). Non è possibile specificarli! ⁵⁴

Astrazione di tipo

Una *astrazione di tipo* (o *tipo astratto di dato* o, ancora più semplicemente, *tipo astratto*) ha un corpo costituito da una espressione di tipo. Quando è valutata, l'astrazione di tipo stabilisce sia una rappresentazione per un insieme di valori e sia le operazioni ad essi applicabili.

Analogamente a quanto detto per le altre astrazioni, l'astrazione di tipo potrà essere specificata come segue:

type I(FP₁; ...; FP_n) is T

dove **I** è un identificatore del nuovo tipo, **FP₁; ...; FP_n** sono parametri formali, e **T** è una espressione di tipo che specificherà la rappresentazione dei dati di tipo **I** e le operazioni ad esso applicabili.

Astrazione di tipo

Analizziamo le astrazioni di tipo che alcuni linguaggi di programmazione (come C e Pascal) consentono di definire.

```
type complex = record  
           Re: real;  
           Im: real  
end
```

consente di:

1. stabilire che `complex` è un identificatore di tipo;
2. associare una rappresentazione a `complex` espressa mediante tipi concreti già disponibili nel linguaggio.

Le operazioni associate al nuovo tipo `complex` sono tutte quelle che il linguaggio ha già previsto per il tipo `record` (e.g., assegnazione e selezione di campi).

Astrazione di tipo

I **limiti** di questa astrazione di tipo *à la* Pascal sono:

1. Il programmatore **non** può definire nuovi operatori specifici da **associare** al tipo.
2. **Violazione del requisito di protezione**: l'utilizzatore è consapevole della rappresentazione del tipo `complex` (sa che è un record e quali sono i campi) ed è in grado di operare mediante operatori non specifici del dato.
3. **L'astrazione di tipo non è parametrizzata** (la comunicazione con il contesto esterno non è ammessa).

Di seguito si mostreranno le necessarie estensioni del linguaggio per superare questi limiti.

Astrazione di tipo

Problema: Il programmatore **non** può definire nuovi operatori specifici da **associare** al tipo.

Il problema può essere risolto mediante un costrutto di programmazione che permette di **incapsulare**

1. rappresentazioni del dato,
2. operatori leciti.

QUI SI POSSONO UTILIZZARE
I MODIFICATORI DI VISIBILITÀ

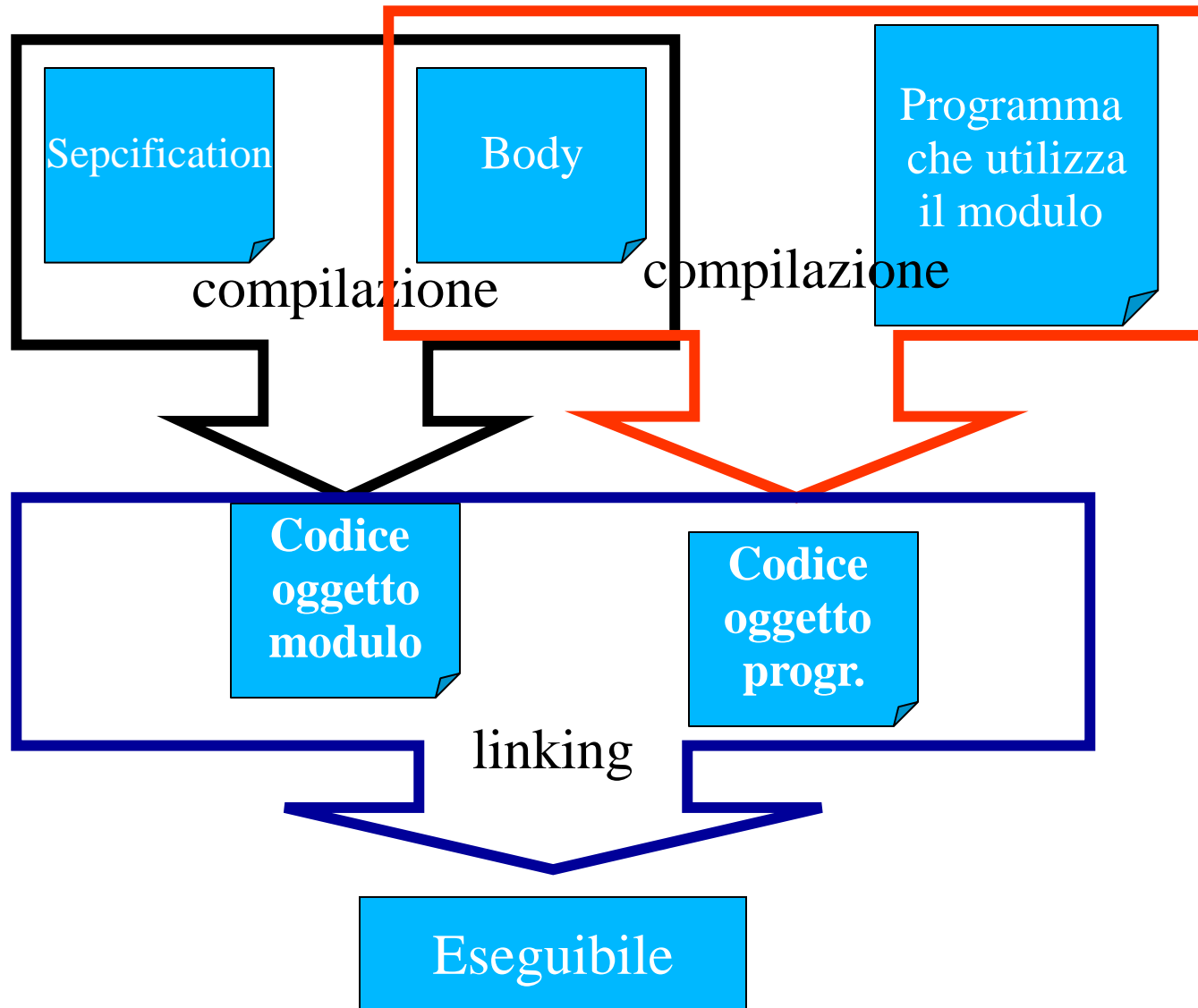


Questo costrutto di programmazione è il **package**, ovvero

un gruppo di componenti dichiarate, come tipi, costanti, variabili, funzioni e persino (sotto) moduli.

Introduciamo quindi un costrutto **package** per supportare l'incapsulamento.

I package: Distribuzione in file





Il Package in Ada

In Ada il modulo è chiamato **package** e si compone di due parti:

1. una **specification**, che interagisce con l'ambiente esterno: contiene le dichiarazioni di tipi, costanti, procedure e funzioni;
2. un **body**, che contiene, fra l'altro, l'implementazione di procedure e funzioni dichiarate nella **specification**, ed eventualmente una routine di inizializzazione del package.

Il Package in Ada

A sua volta la *specification* si articola in due sottoparti:

1. **visible**: le entità dichiarate in questa parte possono essere rese note ad altre unità di programma per mezzo della clausola `use`;
2. **private**: le entità dichiarate in questa parte non possono essere né esportate e né dichiarate nel corpo.

Il Package in Ada

La parte di specifica inizia con la parola chiave **package** seguita dall'identificatore del package e da **is**; seguono poi le *dichiarazioni* delle entità visibili e private.

Esempio:

```
package Type_complex is
```

```
  type Complex is
```

```
    record
```

```
      RL, IM: Real;
```

```
    end record;
```

```
  l: constant Complex := (0.0, 1.0)
```

```
  function "+"(x,y: Complex) return Complex
```

```
  ...
```

```
end Type_complex;
```

Pubblica il tipo ma anche la sua implementazione, la quale non è definita private

Il Package in Ada

Questo è un esempio di specifica di un package che realizza il tipo astratto **Complex**. In questo non c'è una parte privata nella specifica, sicché un programma che fa uso di questo package può operare su variabili complesse mediante espressioni come la seguente:

C.IM := C.IM + 1.0;

invece di ricorrere alla forma più astratta:

C := C + I;

Per evitare il problema possiamo nascondere la struttura del tipo **Complex** nella parte privata.

Il Package in Ada

```
package Type_complex is  
  type Complex is private;  
  I: constant Complex;  
  function "+"(x,y: Complex) return Complex;  
  ...  
private  
  type Complex is          record  
                           RL, IM: Real;  
                           end record;  
  I: constant Complex := (0.0, 1.0)  
end Type_complex;
```

N.B.: avendo dichiarato il tipo **Complex** nella parte privata non è più possibile inizializzare la costante **I** nella parte pubblica, in quanto non è ancora nota la rappresentazione di **Complex**.

Il Package in Ada

Anche il corpo del package incomincia con la parola **package** seguita però dalla parola **body**, dall'identificatore e da **is**. Nel corpo sono fornite le implementazioni delle procedure e funzioni dichiarate nella corrispondente specification.

```
package body Type_complex is  
function "+"(x,y: in Complex) return Complex is  
    begin  
        return(x.RL+y.RL, x.IM+y.IM);  
end "+";  
...  
end Type_complex;
```

Il Package in Ada

Ovviamente la struttura del tipo **Complex** è visibile alla parte body del package, quindi si potrà accedere ai campi **RL** e **IM** dei record **Complex**.

Il package potrà essere utilizzato come segue:

```
with Type_complex; use Type_complex;
```

```
procedure main is
```

```
  cpx1, cpx2: Complex;
```

```
begin
```

```
  ...
```

```
  cpx1 := cpx2 + 1;
```

```
  ...
```

```
end main;
```

Il Package in Ada

NO VARIABILI GLOBALI

Si osservi che sia nella specifica e sia nel corpo di `Type_complex` non c'è alcuna dichiarazione di variabili esterne a procedure e funzioni. Ciò vuol dire che questo package non è dotato di uno stato locale, cioè *non definisce un oggetto*.

Per questa ragione il corpo del package non necessita di un “main”: non si deve inizializzare un oggetto.

Il package *può in ogni caso avere un proprio main*. Esso verrà specificato dopo le varie procedure e funzioni e sarà compreso fra un `begin` e l'`end` del package.

Il Package in Ada

Esempio:

```
with Simple_io; use Simple_io;  
package body Type_complex is  
function "+"(x,y: in Complex) return Complex is  
    begin  
        return(x.RL+y.RL, x.IM+y.IM);  
end "+";  
...  
begin  
    put("Main of package Type_complex ");  
end Type_complex;
```

L'esecuzione del main del package avverrà al momento in cui si importa il package mediante la clausola `use`. Si vedrà quindi visualizzare la frase "Main of package Type_complex"

Il Package in Ada

Esempio: astrazione di tipo per realizzare il dato astratto Pila

package Type_stack **is**

nome package arbitrario
nome dato astratto da rispettare!!!!

type Stack **is private**

procedure push(s:in out Stack; x:in Integer);

Manca il costruttore perchè è implicito
nella dichiarazione della variabile

procedure pop(s:in out Stack);

procedure top(s:in Stack; x:out Integer);

function empty(s:in Stack) **return** Boolean;

private

max: **constant**:= 100;

type Stack **is limited record**

limited perchè mancano gli operatori
uguale e assegnamento

st:array(1..max) **of** Integer;

top: Integer **range** 0..max := 0;

end record;

end Type_stack;

```
package body Type_stack is
  procedure push(s:in out Stack; x:in Integer) is
  begin
    s.top := s.top+1;
    s.st(s.top) := x;
  end push;
  procedure pop(s:in out Stack) is
  begin
    s.top := s.top - 1;
  end pop;
  procedure top(s:in Stack; x:out Integer) is
  begin
    x := s.st(s.top);
  end top;
  function empty(s:in Stack) return Boolean is;
  begin
    return(s.top=0);
  end empty;
end Type_stack;
```

Il Package in Ada



- Le specifiche del tipo astratto *Pila* prevedono anche un costruttore *CreaPila*, che non ha un corrispondente in questa definizione del tipo astratto *Stack*.
- La ragione è che si stanno utilizzando costruttori impliciti forniti nel linguaggio Ada, come la dichiarazione di una variabile di tipo *Stack*.
- Tuttavia se un costruttore dovesse essere parametrizzato (come quello di *Conto Con Fido*) sarà necessario prevedere un metodo.

Il Package in Ada

In Ada, definendo un tipo come *private* è possibile applicare su istanze di quel tipo tutti i metodi definiti nella parte pubblica della specifica, ma anche effettuare *assegnazioni* e *confronti di (dis-)eguaglianza*.

Queste operazioni che il compilatore offre ‘gratuitamente’ per un tipo privato devono necessariamente essere definite in modo generale, indipendentemente da come il tipo è poi definito.

Quindi saranno implementate semplicemente *copiando o confrontando byte a byte* le aree di memoria riservate a due dati dello stesso tipo dichiarato come privato.

Il Package in Ada

Problemi:



- 1) queste tre operazioni (assegnazione $:=$, confronto per eguaglianza $=$, confronto per disequaglianza \neq) potrebbero non far parte della specifica di un dato astratto. Quindi potrebbe non essere corrette offrirle all'utilizzatore del tipo dichiarato come privato.
- 2) la semantica delle operazioni potrebbe essere diversa da quella stabilita dal compilatore. Facciamo un esempio ...

Il Package in Ada

```
with Type_stack, use Type_stack,  
procedure main is  
    st1, st2: Stack;  
    cmp: Boolean;  
begin  
    push( st1, 1);  
    push( st2, 1);  
    push( st1, 2);  
    pop(st1);  
    cmp := st1 = st2  
end main;
```

Il valore di `cmp` è `False`. Infatti i corrispondenti campi `top` dei record `st1` ed `st2` sarebbero identici, mentre non avrebbero gli stessi valori i corrispondenti campi `st`. Eppure i due stack sarebbero identici secondo la specifica algebrica di $equal(l,m)$ data nel progetto.

Il Package in Ada

Il tipo **limited private**

Per evitare tutto ciò, è sufficiente dichiarare il tipo come **limited private**, che inibisce l'uso delle operazioni di assegnazione e confronto offerte per default dal compilatore.

package Type_stack **is**

type Stack **is** **limited private**

procedure push(s:**in out** Stack; x:**in** Integer);

procedure pop(s:**in out** Stack);

procedure top(s:**in** Stack; x:**out** Integer);

function empty(s:**in** Stack) **return** Boolean;

...

Il Package in Ada

In Ada è anche possibile definire **oggetti**, cioè moduli dotati di stato locale.

Esempio: oggetto stack.

```
package Stack is  
    procedure push(x:in Integer);  
    procedure pop;  
    procedure top(x:out Integer);  
    function empty return Boolean;  
end Stack;
```

package body Stack is

max: **constant**:= 100;

type Table **is** array(1..max) **of** Integer;

st: Table;

top: Integer **range** 0..max := 0;

procedure push(x:in Integer) **is**
begin

top := top+1;

st(top) := x;

end push;

procedure pop **is**

begin

top := top - 1;

end pop;

procedure top(x:out Integer) **is**
begin

x := st(top);

end top;

function empty **return** Boolean **is**;

begin

return(top=0);

end empty;

end Stack;

Il Package in Ada

Si potrà quindi utilizzare l'oggetto Stack nel modo seguente:

```
with Stack; use Stack;  
procedure main is  
begin  
....  
    push(1);  
    push(2);  
    pop  
    if empty then push(1);  
...  
end main;
```

Classi di oggetti

In generale, un oggetto è un insieme di variabili interne ad un modulo e manipolabili esternamente solo mediante gli operatori (pubblici) definiti nel modulo stesso.

Da quanto visto finora, per poter definire più oggetti “simili” o “dello stesso tipo”, cioè con medesima rappresentazione e stesso insieme di operatori, si è costretti a definire tanti moduli quanti sono gli oggetti che si vogliono usare nel programma. Tutti questi moduli differiranno solo per l’identificatore del modulo (*identificatore dell’oggetto*).

Per evitare l’inconveniente di dover duplicare un modulo si può pensare di definire un *package generico* che identifica una *classe* di oggetti simili. I singoli oggetti sono poi ottenuti con il meccanismo della *istanziatura* della classe.



Generic Package in Ada

In Ada un package che specifica e implementa un singolo oggetto può essere facilmente trasformato in un *generic package*, che definisce una classe di oggetti, premettendo la parola **generic** alla dichiarazione del modulo.

generic

package STACK

procedure PUSH(X:in INTEGER) ;

In questo modo si definisce solo una *matrice* degli oggetti da creare. Per ottenere i singoli oggetti dobbiamo **istanziare** il generic package:

package ST1 is new STACK

package ST2 is new STACK

Generic Package in Ada

Queste due dichiarazioni sono processate in fase di **precompilazione**. In particolare, per ogni occorrenza di istanziamento,

- Si sostituisce la stringa dell'istanziamento con il comando di importazione di un package avente come nome quello dell'istanza (esempio: **with ST1; use ST1**);
- Si genera un package (non generico) utilizzando il **generic package come matrice**. Esso si ottiene rimuovendo la parola *generic* e sostituendo il nome dell'istanza al posto del nome del package. Il package è poi compilato separatamente.

Nell'esempio specifico, la precompilazione genera **due package distinti**, che variano solo nel nome del package, cioè nell'*identificatore dell'oggetto*.

Generic Package in Ada

Osservazioni:

- Il package generico **non è compilabile separatamente** in quanto il nome del package non identifica un oggetto;
- La generazione di package distinti, uno per ogni istanziatura, comporta la creazione di molteplici copie dello stesso codice (**inefficienza in spazio**).
- La creazione dei legami (binding) al compile-time garantisce **l'efficienza in tempo**, poiché non è necessario effettuare computazioni di legami al run-time.
- L'ambiguità dovuta alle molteplici occorrenze di metodi con stesso identificatore richiede il ricorso, nel programma utilizzatore, alla notazione **⟨identificatore di oggetto⟩.⟨nome metodo⟩** (ad esempio, **ST1.push(10)**)

Astrazione della dichiarazione di modulo

- La precedente definizione di una **classe** corrisponde a una particolare forma di astrazione, quella della classe sintattica '*dichiarazione di un modulo*'.
- L'operazione di istanziiazione corrisponde alla invocazione di questa astrazione ed ha l'effetto di 'creare legami' (binding). In particolare, si crea un legame fra
 - Identificatore dell'oggetto
 - Nome della classe (cioè nome del modulo generico)
- Pertanto la definizione di una classe corrisponde a una particolare forma di **astrazione generica**, cioè di astrazione applicata alla classe sintattica dichiarazione.

Astrazione della dichiarazione di modulo

- Quando si affronterà l'argomento dell'astrazione generica in modo sistematico, si osserverà che è possibile astrarre anche su altre dichiarazioni (per esempio, funzioni e procedure), oltre quella di modulo.

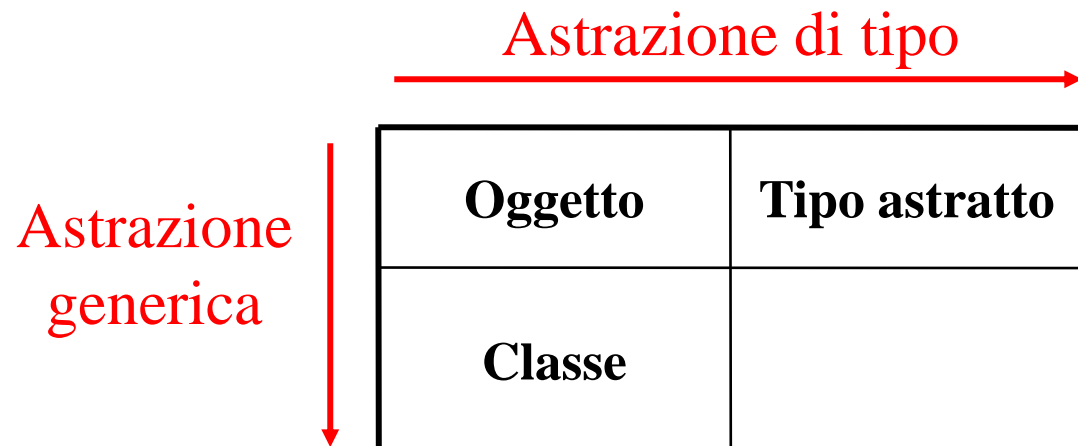


Riflessione: si è già visto che un modulo può essere usato per incapsulare un **tipo astratto** con gli operatori leciti. Ha senso definire *generic* un modulo di questa fatta e poi istanziarlo?

Tipi astratti o classi?

Riepilogando, se in fase di progettazione si identifica l'esigenza di disporre di un dato astratto, in fase realizzativa si può:

- 1) **Definire un oggetto**: la scelta è appropriata nel caso in cui si necessita di una sola occorrenza del dato astratto;
- 2) **Definire un tipo astratto**: l'astrazione riguarda la classe sintattica tipo;
- 3) **Definire una classe**: l'astrazione riguarda la dichiarazione di un modulo (dotato di stato locale).



Tipi astratti o classi?

In tutti i casi la rappresentazione del dato astratto viene nascosta e la manipolazione dei valori è resa possibile solo mediante operazioni fornite allo scopo.

Tuttavia ci sono delle **differenze** fra tipo astratto e classe di oggetti ...

Tipi astratti o classi?

- *Sintattica*: nel tipo astratto gli operatori hanno **un parametro in più**, relativo proprio al tipo che si sta definendo.

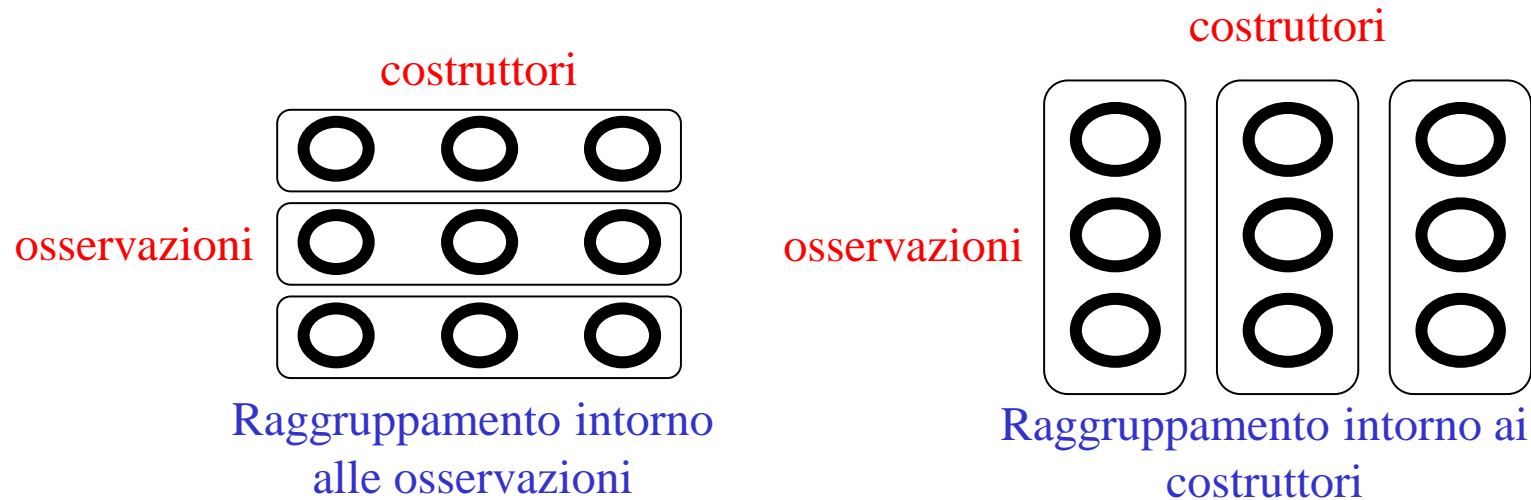
Esempio: nel tipo astratto Stack, gli operatori POP e PUSH hanno un parametro in più di tipo Stack che gli operatori definiti per la classe Stack non hanno.

- *Realizzativa*: nel caso del tipo astratto gli operatori sono definiti una sola volta, mentre nel caso della classe gli operatori sono definiti più volte, tante quante sono le istanze. Le diverse copie degli operatori agiranno su diversi dati (gli oggetti) in memoria centrale.

Tipi astratti o classi?

- **Concettuale:** richiamando la suddivisione delle operazioni su un dato astratto in osservazioni e costruttori (vedi specifiche algebriche), si può dire che:
 - il tipo astratto è **organizzato intorno alle osservazioni**. Ogni osservazione è implementata come una operazione su una rappresentazione concreta derivata dai costruttori. Anche i costruttori sono implementati come operazioni che creano valori. La rappresentazione è **condivisa** dalle operazioni, ma è nascosta ai fruitori del tipo astratto.
 - La classe è **organizzata intorno ai costruttori**. Le osservazioni diventano metodi dei valori. Un oggetto è definito dalla combinazione di tutte le osservazioni possibili su di esso.

Tipi astratti o classi?



In breve, un tipo astratto di dato può essere inteso **come un insieme con operazioni** (come un'algebra, insomma) mentre le classi sono **insiemi di operazioni**.

Vantaggi del tipo astratto

- Nei linguaggi imperativi, i valori di un tipo astratto vengono trattati alla stregua dei valori di un tipo concreto, cioè sono cittadini di **prima classe**. Al contrario i valori rappresentati mediante oggetti sono trattati come cittadini di **terza classe** in quanto:
 - una procedura non può restituire l'istanza di un generic package;
 - non è possibile creare dinamicamente degli oggetti (le istanze sono stabilite al momento della compilazione).
- I tipi astratti sono utili in tutti i paradigmi di programmazione, mentre gli oggetti, essendo variabili aggiornabili, si adattano bene solo a un paradigma di programmazione side-effecting.
- La notazione usata per chiamare un'operazione di un tipo astratto è più naturale perché valori e variabili del tipo astratto sono argomenti espliciti.

Svantaggi del tipo astratto

- *Scarsa estendibilità*: l'aggiunta di un nuovo costruttore comporta dei cambiamenti intrusivi nelle implementazioni esistenti degli operatori.

Ogni operatore dovrà essere opportunamente rivisto in modo da prevedere il trattamento di rappresentazioni ottenute con nuovi costruttori.

Esempio: si vuole implementare il dato astratto *geometricShape* la cui specifica algebrica è fornita di seguito:

<i>osservazioni</i>	<i>Costruttore di g</i>	
	<i>square(x)</i>	<i>circle(r)</i>
<i>area(g)</i>	x^2	πr^2

Svantaggi del tipo astratto

```
package geometric shape_type
```

```
type geometricShape is limited private
```

```
function square(x: real): geometricShape
```

```
function circle(x: real): geometricShape
```

Aggiungi in dopo :

```
function area(g: geometricShape): real
```

```
private
```

```
type geometricShape = record
```

```
shape: char;
```

```
value: real
```

```
end
```

```
end geometric shape_type
```

Svantaggi del tipo astratto

package body geometric_shape_type

function *square(x: real): geometricShape*

var g: geometricShape

begin

g.shape := 's';

g.value := x;

return g

end

function *circle(x: real): geometricShape*

var g: geometricShape

begin

g.shape := 'c';

g.value := x;

return g

end

function *area(g: geometricShape): real*

begin

if g.shape = 's' then return g.value * g.value

else return 3.14 * g.value * g.value;

end

end geometric_shape_type

Svantaggi del tipo astratto

Realizzando il dato astratto mediante **classi** si possono definire due moduli generici, uno per ogni forma geometrica (**o costruttore**).

```
generic package Circle
  function area(): real;
  procedure init(real);
end Circle
Package body Circle
  var raggio:real;
  procedure init(x:real)
  begin
    raggio := x
  end;
  function area():real
  begin
    return 3.14*raggio*raggio
  end;
end Circle.
```

```
generic package Square
  function area(): real;
  procedure init(real);
end Square
package body Square
  var lato:real;
  procedure init(x:real)
  begin
    lato := x
  end;
  function area():real
  begin
    return lato*lato
  end;
end Square.
```

Per utilizzare una forma geometrica si deve istanziare una delle due classi e invocare il metodo *init*.

Svantaggi del tipo astratto

Cosa succede se si estende la specifica del dato astratto in modo da considerare anche i rettangoli?

<i>osservazioni</i>	<i>Costruttore di g</i>		
	<i>square(x)</i>	<i>circle(r)</i>	<i>rectangle(l,m)</i>
<i>area(g)</i>	x^2	πr^2	$l \cdot m$

Se abbiamo specificato un **tipo astratto**, siamo stati costretti a cambiare la rappresentazione in modo da memorizzare due valori (per i due lati del rettangolo) e non uno.

```
type geometricShape = record
```

```
  shape: char;
```

```
  value: real;
```

```
  value2: real
```

```
end
```

Svantaggi del tipo astratto

Inoltre dobbiamo aggiungere l'opportuno costruttore ...

```
function rectangle(x,y: real): geometricShape
```

```
var g: geometricShape
```

```
begin
```

```
  g.shape := 'r';
```

```
  g.value := x;
```

```
  g.value2 := y;
```

```
  return g
```

```
end
```

... e dobbiamo modificare la
anche la funzione area ☹

```
function area(g: geometricShape): real
```

```
begin
```

```
  if g.shape = 's' then return g.value * g.value
```

```
    else if g.shape='c' then return 3.14 * g.value * g.value
```

```
      else return g.value * g.value2
```

```
end
```

Sempre supponendo di disporre del codice sorgente del modulo! 96

Svantaggi del tipo astratto

Diversamente, avendo realizzato il dato astratto mediante **classi** basta aggiungere un'altra classe.

```
generic package Rectangle
  function area(): real;
  procedure init(real,real);
end
package body Rectangle
  var base, altezza:real;
  procedure init(x,y:real)
  begin
    base := x;
    altezza := y
  end;
  function area():real
  begin
    return base*altezza
  end;
end Rectangle.
```

Per utilizzare un rettangolo si dovrà istanziare questa classe e invocare il metodo *init*.

Nella programmazione orientata a oggetti, per evitare la riscrittura di codice comune ad altre classi già definite è possibile ricorrere ai meccanismi di **ereditarietà** tra le classi.

Astrazione generica

Il concetto di classe di oggetti va inquadrato in un tema più generale, quello dell'**astrazione generica**.

Il principio di astrazione suggerisce che si può astrarre anche sulla classe sintattica **dichiarazione**, in quanto essa sottintende una computazione. In particolare la valutazione di una dichiarazione comporta la creazione di **legami** (*bindings*).

Un'**astrazione generica** è un'astrazione su una dichiarazione, pertanto il corpo della dichiarazione di una astrazione generica è a sua volta una dichiarazione. La chiamata di un'astrazione generica è detta **istanziamento** e produce dei legami elaborando la dichiarazione contenuta nel corpo dell'astrazione generica.

Astrazione generica

Analogamente a quanto detto per le altre astrazioni, l'astrazione generica potrà essere specificata come segue:

generic $I(FP_1; \dots; FP_n)$ is D

dove **I** è un identificatore dell'astrazione generica, **$FP_1; \dots; FP_n$** sono parametri formali, e **D** è una dichiarazione che, quando elaborata, produrrà dei legami. **D** funge da *matrice* dalla quale ricavare le dichiarazioni per istanziazione.

Una dichiarazione D può essere:

- La dichiarazione di un tipo;
- La dichiarazione di un modulo;
- La dichiarazione di una funzione;
- La dichiarazione di una procedura;
- ...

Astrazione generica

Occorre pensare a dei meccanismi per poter distinguere le diverse dichiarazioni che si ottengono per istanziamento. Un modo è quello di specificare un diverso identificatore all'atto della istanziamento, come nel seguente esempio:

A instantiation of I;

Così, la seguente dichiarazione generica

generic type RxR is

type RxR = Record

x: real;

y: real

end;

potrà essere utilizzata come matrice per generare le dichiarazioni per i tipi *Point2D* e *Complex*:

Point2D instantiation of type RxR

Complex instantiation of type RxR

Astrazione generica

Si è visto un utilizzo dell'astrazione generica nella dichiarazione di moduli dotati di stato locale.

Mediante l'operazione di istanziamento si ottengono diverse copie dell'oggetto che differiscono solo per il nome dell'identificatore.

L'astrazione generica di un oggetto corrisponde al concetto di classe.

Generic Package in Ada

Il *generic package* di Ada è una esemplificazione di astrazione generica. L'espressione:

```
package ST1 is new STACK
```

è un esempio di istanziamento generica.

Astrazione generica

Le astrazioni generiche, come qualsiasi altra astrazione, possono essere parametrizzate.

Esempio: Nel seguente esempio, viene definita una classe coda in Ada. La variabile **items** è un array di caratteri. Al fine di svincolare la definizione di classe da particolari costanti legate all'applicazione si dota l'astrazione generica del *parametro formale*, **capacity**, che è utilizzato per dimensionare l'array. L'istanziamento deve consentire di specificare il parametro effettivo, che sarà un valore da associare al parametro formale. In Ada l'istanziamento sarà specificata come segue:

```
package line_buffer is new queue_class(120)103
```

Astrazione generica

generic

capacity: Positive;

package queue_class is

 procedure append(newitem: in Character);

 procedure remove(olditem: out Character);

end queue_class;

package body queue_class is

 items: array(1..**capacity**) of Character;

 size, front, rear: Integer range 0..**capacity**;

 procedure append(newitem: in Character) is

 ... ;

 procedure remove(olditem: out Character) is

 ... ;

begin

...

end queue_class;

Astrazione generica

In principio si può applicare l'astrazione generica a qualunque dichiarazione, incluso le procedure e le funzioni. Ad esempio, si potrebbe dichiarare una procedura **T_swap** per scambiare dati di un tipo **T** **predefinito**:

```
generic
procedure T_swap(a,b: in out T) ;
procedure T_swap(a,b: in out T) is
    tmp: T;
begin
    tmp :=a; a:=b; b:=tmp;
end T_swap;
```

e ottenere diverse copie di essa per istanziiazione:

```
procedure swap1 is new T_swap
procedure swap2 is new T_swap
```

Astrazione generica

In realtà è poco utile disporre di due funzioni identiche ma di nome diverso. Diversa sarebbe la situazione se potessimo dichiarare una generica procedure **T_swap** che opera su dati di tipo **T** qualunque, e potessimo specificare il tipo al momento dell'istanziamento. Per ottenere questo risultato necessitiamo di una particolare classe di **parametri**, quelli **di tipo**.

Esempio (in Ada):

```
generic
  type T is private;
procedure T_swap(a,b: in out T) ;
procedure T_swap(a,b: in out T) is
  tmp: T;
begin
  tmp :=a; a:=b; b:=tmp;
end T_swap;
```

Astrazione generica

La clausola *generic* introduce un **parametro di tipo** e la dichiarazione che segue introduce la matrice di una procedura che scambia due dati di un tipo **T** generico. Le procedure effettive sono ottenute istanziando la procedura generica con i parametri di tipo effettivi da sostituire a **T**.

Esempio:

```
procedure int_swap is new T_swap(INTEGER) ;
```

```
procedure str_swap is new T_swap(STRING) ;
```

Assumendo che *i* e *j* sono variabili di tipo **INTEGER** e che *s* e *t* sono variabili di tipo **STRING** allora:

```
int_swap(i, j) ;
```

```
str_swap(s, t) ;
```

corretto

```
int_swap(i, s) ;
```

```
str_swap(s, j) ;
```

```
str_swap(i, j) ;
```

Astrazione generica

In questo modo si è:

- Svincolato la definizione dello scambio di due elementi da un fattore marginale, come il tipo degli elementi da scambiare;
- Garantito comunque il **controllo statico dei tipi** fra parametri formali e parametri effettivi delle diverse procedure ottenute, e fra sorgente e destinazione di una assegnazione.

L'uso dei parametri di tipo in astrazioni generiche offre un buon **compromesso** fra necessità di dover effettuare il **controllo statico dei tipi** e desiderio di definire componenti software riutilizzabili.

Astrazione generica

I parametri di tipo possono essere utilizzati anche quando si definiscono delle classi, come indicato in questo esempio (in Ada)

generic

max: Positive;

type ITEM is private;

package Stack is

procedure push(x:in ITEM);

procedure pop;

procedure top(x:out ITEM);

function empty **return** Boolean;

end Stack;

package body Stack is

type Table **is array**(1..max) **of** ITEM;

st: Table;

top: Integer **range** 0..max := 0;

Astrazione generica

```
procedure push(x:in ITEM) is  
begin
```

```
    top := top+1;  
    st(top) := x;
```

```
end push;
```

```
procedure pop is  
begin
```

```
    top := top - 1;
```

```
end pop;
```

```
procedure top(x:out ITEM) is  
begin
```

```
    x := st(top);
```

```
end top;
```

```
function empty return Boolean is  
begin
```

```
    ...
```

Astrazione generica

In questo caso per creare i singoli oggetti scriveremo:
declare

```
package STACK_INT is new STACK(10,INTEGER);  
use STACK_INT;  
package STACK_REAL is new STACK(10,REAL);  
use STACK_REAL;  
A: REAL; B:INTEGER;
```

begin

```
push(12);push(15.0);top(B);top(A)
```

end

Si osservi che non è necessario utilizzare la notazione puntata:

```
STACK_INT.push(10)
```

```
STACK_REAL.push(15.0)
```

in quanto `push` e `top` sono differenziate dal contesto (tipo di parametro effettivo passato).

Questo è un caso di *overloading* come si chiarirà meglio in seguito.¹¹¹

Astrazione generica

L'astrazione generica è quindi di supporto all'astrazione dati, in quanto permette di definire delle classi che sono invarianti ad alcuni tipi di dati necessari per definirle.

Non solo. L'astrazione generica, mediante i parametri di tipo, è applicabile a tipi astratti che possono essere così ugualmente svincolati dalla necessità di specificare il tipo degli elementi sui quali operare.

Esempio:

Si consideri il problema di definire dei tipi astratti per una applicazione che usa:

- 1) Stack di interi;
- 2) Stack di reali;
- 3) Stack di un tipo astratto *point3D* utilizzato per rappresentare i punti di uno spazio tridimensionale.

Cosa fare?

Astrazione generica

Una alternativa sarebbe quella di scrivere una definizione separata per ciascuno dei tre tipi.

Svantaggi:

1. **Codice duplicato** in quanto plausibilmente simile per tutte le definizioni (differisce solo nelle parti in cui si fa riferimento ai singoli elementi dello stack).
2. **Sforzo di programmazione ridondante.**
3. **Manutenzione complicata** poiché le modifiche, come l'aggiunta di un nuovo operatore, vanno plausibilmente effettuate in tutte le versioni.

Una alternativa sarebbe quella di *separare le proprietà di uno stack dalle proprietà dei loro elementi.*

Come? Utilizzando i parametri di tipo.

Astrazione generica

generic

MAX: Positive;

type ITEM is private;

package STACKS is

 type STACK is limited private;

 procedure PUSH(S:in out STACK; E:in ITEM) ;

 procedure POP(S: in out STACK; E: out ITEM) ;

private

 type STACK is

 record

 ST: array(1..MAX) of ITEM;

 TOP: integer range 0..MAX;

 end record;

end;

Astrazione generica

```
package body STACKS is
    procedure PUSH(S: in out STACK; E: in ITEM) ;
begin
    S.TOP := S.TOP - 1;
    S.ST(S.TOP) := E;
end PUSH;
procedure POP(S: in out STACK; E: out ITEM) ;
begin
    E := S.ST(S.TOP) ;
    S.TOP := S.TOP - 1;
end POP;
end STACKS
```

In questo modo si è definito un ***tipo astratto generico*** STACK.

Astrazione generica

I diversi stack richiesti dall'applicazione sono ottenuti per istanziiazione:

```
declare
    package MY_STACK is new STACKS(100, REAL) ;
    use MY_STACK;
    x: STACK; I:REAL;
begin
    push(x, 175.0) ;
    pop(x, I)
end
```

Astrazione generica

Se un'astrazione è parametrizzata rispetto a un valore, possiamo usare l'argomento valore anche se non sappiamo nulla al di fuori del suo tipo. Analogamente se un'astrazione è parametrizzata rispetto a una variabile, possiamo ispezionare e aggiornare la variabile argomento anche se non sappiamo nulla oltre il suo tipo.

Ma quando si parametrizza rispetto al tipo la situazione cambia. Nell'esempio di **T_swap** avevamo le seguenti assegnazioni:

tmp := a; a := b; b := tmp;

ma chi garantisce che l'assegnazione sia una operazione valida per i dati di tipo **T**?

Astrazione generica

In Ada l'espressione

type T is private;

sottintende che l'assegnazione e i predicati $=$ e \neq sono operazioni valide per il tipo effettivo denotato da T. Per questo, tutti gli esempi visti precedentemente non creavano problemi.

Riflessione: Se T fosse stato definito come **limited private** ?

Astrazione generica in C++

In C++ l'astrazione generica è supportata mediante i *template*.

Un template è del codice generico dotato di parametri che possono assumere valori specifici al momento della compilazione (compile-time).

Ad esempio, piuttosto che scrivere due classi, ListofInts e ListofStrings, si potrebbe scrivere una singola classe template:

```
template<class T> class List
```

dove il parametro di tipo (*class*) T del template può essere rimpiazzato da un qualunque tipo quando il codice è compilato.

Ciò è ottenuto mediante una istanziiazione del template:

```
List<int> l1;
```

```
List<string> h2;
```

```
List<int> l3;
```

Astrazione generica in C++

Esempio:

```
template<class T> class inutile {  
    T x;  
public:  
    T getx() {return x;}  
    void setx(T y) { x = y};  
};
```

Possiamo creare due istanziazioni della classe template, usando in questo caso l'istanziatura esplicita:

```
template class inutile<int>;  
template class inutile<char>;
```

Il compilatore genererà due definizioni di classi, una per ogni istanziazione.

Astrazione generica in C++

Esempio:

```
class inutile<int> {  
    int x;  
public:  
    int getx() {return bar;}  
    void setx(int y) { x = y};  
};  
class inutile<char> {  
    char x;  
public:  
    char getx() {return bar;}  
    void setx(char y) { x = y};  
};
```

In generale ogni istanziazione di un template produce una copia del codice template.

La copia è creata in fase di precompilazione.

Riflessione: Che succede in questo caso?

```
template class inutile<int>;  
template class inutile<int>;
```

Astrazione generica in C++

Si distinguono due categorie di template in C++:

1. *Template di classe*: definisce la struttura e le operazioni per un insieme illimitato di tipi correlati.

Esempio:

Un singolo template di classe **Stack** potrebbe fornire una definizione comune per una pila di `int`, di `float`, e così via. Nella seguente dichiarazione della classe **Stack** il parametro di tipo `T` denota il tipo degli elementi contenuti in una pila.

Astrazione generica in C++

```
template<class T> class Stack{  
    int top;  
    int size;  
    T* elements;  
public:  
    Stack(int n) {size=n; elements=new T(size); top =0;}  
    ~Stack() {delete[] elements;}  
    void push(T a)    {top++; elements[top]=a;}  
    T pop() {top--; return elements[top+1];}  
};
```

Quando Stack è usato come nome di tipo, esso dev'essere accompagnato da un tipo come parametro esplicito.

Stack<int> s(99); ← stack di interi di dim. 99

Stack<char> t(80); ← stack di caratteri di dim. 80

Astrazione generica in C++

2. Template di funzione: definisce un insieme illimitato di funzioni correlate.

Esempio:

Una famiglia di funzioni di ordinamento potrebbe essere dichiarata in questo modo:

```
template<class T> void sort (vector<T>);
```

Una funzione generata da una template di funzione è chiamata **funzione template**.

```
vector <complex> cv(100);
```

```
vector <int> ci(200);
```

```
void f(vector <complex> &cv, vector<int> &ci)
```

```
{    sort(cv);
```

```
    sort(ci);
```

```
    sort(cv);
```

```
}
```

Astrazione generica in C++

Si osservi che in C++ **non è necessaria l'istanziazione esplicita** dei template.

Nell'esempio, saranno istanziate due funzioni:

`sort(vector<complex>)` e `sort(vector<int>)`.

Le diverse istanze sono generate sulla base dei tipi dei parametri effettivi delle chiamate a funzioni generiche.

Come si vede c'è ambiguità nella invocazione della funzione `sort`. In particolare, `sort(cv)` si riferisce a `sort(vector<complex>)` mentre `sort(ci)` si riferisce a `sort(vector<int>)`.

Si ha un caso di **overloading**, ovvero di associazione dello stesso identificatore di funzione a realizzazioni differenti. Il compilatore risolve questa ambiguità sulla base del tipo degli argomenti effettivi.

Template vs. Generics

Nonostante la similarità sintattica, i template del C++ e le Generics di Java sono piuttosto differenti. La differenza è soprattutto nel modo in cui sono trattate dal compilatore.

- Il compilatore C++ genera del codice specifico per ogni istanziazione del template.
- Il compilatore Java introduce dei controlli al compile-time sulle classi generiche in modo da controllare le chiamate ai metodi della classe. Il codice della classe generica non è ricompilato.

Riferimenti bibliografici

M. Shaw

Abstraction Techniques in Modern Programming Languages

IEEE Software, 10-26, October 1984.

D. A. Watt

Programming Language Concepts and Paradigms (cap. 5-6)

Prentice Hall, 1990.

W.R. Cook

Object-Oriented Programming Versus Abstract Data Types

In J.W. de Bakker et al., editor, *Foundations of Object-Oriented Languages*, number 489 in Lecture Notes in Computer Science, pagine 151–178. Springer, 1991.

B. Meyer

Genericity vs. Inheritance

Proceedings OOPSLA '86, pp. 391-405