Il linguaggio C
Overloading e casting

```
main()
{
/*definizioni variabili: */
char y='a'; /*codice(a)=97*/
int       x,X,Y;
unsigned int Z;
float SUM;
double r;

/* parte istruzioni: */
X=27;
Y=343;
Z = X + Y -300;
X = Z / 10 + 23;
Y = (X + Z) / 10 * 10;        /* qui X=30, Y=100, Z=70 */
X = X + 70;
Y  = Y % 10;
Z = Z + X -70;
SUM = Z * 10;       /* X=100, Y=0, Z=100 , SUM=1000.0*/
x=y;       /* char -> int: x=97*/
x=y+x; /*x=194*/
r=y+1.33; /* char -> int -> double*/
x=r; /* coercizione -> troncamento: x=98*/
}
```


OVERLOADING

- In C (come in Pascal, Fortran e molti altri linguaggi) operazioni primitive associate a tipi diversi possono essere denotate con lo stesso simbolo (ad esempio, le operazioni aritmetiche su reali o interi).
- In realtà l'operazione è diversa e può produrre risultati diversi. Per esempio:

```
int X,Y;
se X = 10 e Y = 4;
X/Y vale  ...
```

```
int X; float Y;
se X = 10 e Y = 4.0;
X/Y vale  ...
```

```
float X,Y;
se X = 10.0 e Y = 4.0;
X/Y vale  ...
```

CONVERSIONI DI TIPO

In C è possibile combinare tra di loro operandi di tipo diverso:

- espressioni **omogenee**: tutti gli operandi sono dello stesso tipo
- espressioni **eterogenee**: gli operandi sono di tipi diversi.

- **Regola adottata in C:**
  - sono eseguibili le espressioni eterogenee in cui tutti i tipi referenziati risultano ***compatibili*** (cioè: dopo l'applicazione della **regola automatica di conversione implicita** di tipo del C risultano omogenei).

Conversione implicita di tipo

Data una espressione *x op y.*

1. Ogni variabile di tipo **char** o **short** viene convertita nel tipo **int**;

2. Se dopo l'esecuzione del passo 1 l'espressione è ancora eterogenea, rispetto alla seguente gerarchia

   `int < long < float < double < long double`

   si converte temporaneamente l'operando di tipo *inferiore* al tipo *superiore* (***promotion***);

3. A questo punto l'espressione è **omogenea.** L'operazione specificata puo` essere eseguita se il tipo degli operandi e` compatibile con il tipo dell'operatore. Il risultato è di tipo uguale a quello prodotto dall'operatore effettivamente eseguito.

Conversione implicita di tipo: esempio

```
int x;
char y;
double r;
(x+y) / r
```

Hp: La valutazione dell'espressione procede da sinistra verso destra

- **Passo 1**: `(x+y)`
  - `y` viene convertito nell'intero corrispondente
  - viene applicata la somma tra interi
  - **risultato intero**: *`tmp`*
- **Passo 2**
  - *`tmp`* `/ r` *`tmp`* viene convertito nel double corrispondente
  - viene applicata la divisione tra reali
  - **risultato reale (*double*)**

3

Conversione esplicita di tipo

In C si può forzare la conversione di un dato in un tipo specificato, mediante l'operatore di *cast*:

(<nuovo tipo>) <dato>

➔ il <dato> viene convertito esplicitamente nel <nuovo tipo>.

Esempio:

```
int A, B;
float C;
C=A/(float)B;
```

➔ viene eseguita la divisione tra reali.

Definizione / inizializzazione di variabili di tipo semplice

Definizione di variabili

Tutti gli identificatori di tipo primitivo descritti fin qui possono essere utilizzati per definire variabili.

Ad esempio:

```
char lettera;
int x, y;
unsigned int P;
float media;
```

Inizializzazione di variabili

E` possibile specificare un valore iniziale di una variabile in fase di definizione.

Ad esempio:

```
int x =10;
char y = 'a';
double r = 3.14*2;
int z=x+5;
```


Dichiarazioni e Definizioni

Nella parti dichiarative di un programma C possiamo incontrare:

- definizioni (di variabile, o di funzione)
- dichiarazioni (di tipo o di funzione)

Definizione:

Descrive le proprietà dell'oggetto definito e ne determina l'esistenza.

Ad esempio:

```
int V; /* definizione della variabile intera V */
```

Dichiarazione:

Descrive soltanto delle proprietà di oggetti, che verranno (eventualmente) creati mediante definizione.

Ad esempio: *dichiarazione* di un tipo di dato non primitivo:

```
typedef .... newT; /* newT è un tipo non primitivo*/
```

Dichiarazione di tipo

La dichiarazione di tipo serve per introdurre tipi non primitivi.

```
typedef <descrizione-nuovo-tipo> newT;
```

- si utilizza la parola chiave **`typedef`**.
- la dichiarazione associa ad un tipo di dato non primitivo un identificatore arbitrario (**`newT`**)
- le caratteristiche del nuovo tipo sono indicate in **`<descrizione-nuovo-tipo>`**

L'introduzione di tipi non primitivi aumenta la **leggibilità** e **modificabilità** del programma.

Tipi scalari non primitivi

In C sono possibili dichiarazioni di tipi scalari non primitivi:
- tipi **ridefiniti**
- [tipi **enumerati**] (non li tratteremo)

Tipo ridefinito:
Si ottiene associando un nuovo identificatore a un tipo gia` esistente (primitivo o non).

Sintassi:

```
typedef <id-tipo-esistente> <id-nuovo-tipo> ;
```

Esempio:

```
typedef int MioIntero; /*        MioIntero è un tipo non
                         primitivo che ridefinsce il
                         tipo int*/
MioIntero X;          /* X è di tipo MioIntero */
int Y;                      /* Y è di tipo int */
```

➔ X e Y rappresentano entrambi valori interi (***strutturalmente*** equivalenti), ma ***nominalmente*** sono di tipo diverso.

Istruzione di Assegnamento

- È l'istruzione con cui si modifica il valore di una variabile.

Mediante l'assegnamento, si scrive un nuovo valore nella cella di memoria che rappresenta la variabile specificata.

Sintassi:

<istruzione-assegnamento> ::=
<identificatore-variabile> = <espressione>;

Ad esempio:

```
int A, B;

A=20;
B=A*5; /* B=100 */
```

Compatibilità di tipo ed assegnamento:

In un assegnamento, l'identificatore di variabile e l'espressione devono essere dello stesso tipo (eventualmente, conversione implicita oppure ***coercizione***).

Assegnamento e Coercizione

Facendo riferimento alla gerarchia dei tipi semplici:

```
int < long < float < double < long double
```

consideriamo l'assegnamento:

V=E;

Quando la variabile V e` di un tipo di *rango inferiore* rispetto al tipo dell'espressione E l'assegnamento prevede la conversione forzata (***coercizione***) di E nel tipo di V.

Ad esempio:

```
float k=1.5;
int x;
x=k;   /*il valore di k viene convertito forzatamente nella
   sua parte intera: x assume il valore 1  ! */
```

Esempio:

```
main()
{
/*definizioni variabili: */
char y='a'; /*codice(a)=97*/
int     x,X,Y;
unsigned int Z;
float SUM;
double r;

/* parte istruzioni: */
X=27;
Y=343;
Z = X + Y -300;
X = Z / 10 + 23;
Y = (X + Z) / 10 * 10;        /* qui X=30, Y=100, Z=70 */
X = X + 70;
Y  = Y % 10;
Z = Z + X -70;
SUM = Z * 10;  /* X=100, Y=0, Z=100 , SUM=1000.0*/
x=y;     /* char -> int: x=97*/
x=y+x; /*x=194*/
r=y+1.33; /* char -> int -> double*/
x=r; /* coercizione -> troncamento: x=98*/
}
```


Assegnamento come operatore

Formalmente, **l'istruzione di assegnamento è un'espressione**:

- Il simbolo = è un operatore:
  - → l'istruzione di assegnamento è una espressione
  - → ritorna un valore:
    - il valore restituito è quello assegnato alla variabile a sinistra del simbolo =
    - il tipo del valore restituito è lo stesso tipo della variabile oggetto dell'assegnamento

Ad esempio:

```
int valore=122;
int K, M;

K=valore+100; /* K=222;l'espressione
              produce il
              risultato 222 */
M=(K=K/2)+1;  /* K=111, M=112*/
```

Assegnamento abbreviato

In C sono disponibili operatori che realizzano particolari forme di assegnamento:
- operatori di incremento e decremento
- operatori di assegnamento abbreviato
- operatore sequenziale

- **Operatori di incremento e decremento:**
  Determinano l'incremento/decremento del valore della variabile a cui sono applicati.
  Restituiscono come risultato il valore incrementato/decrementato della variabile a cui sono applicati.

```
int A=10;

A++;          /*equivale a: A=A+1; */
A--;          /*equivale a: A=A-1; */
```

Differenza tra notazione prefissa e postfissa:
- Notazione **Prefissa**: (ad esempio, ++A) significa che l'incremento viene fatto prima dell'impiego del valore di A nella espressione.
- Notazione **Postfissa**: (ad esempio, A++) significa che l'incremento viene effettuato dopo l'impiego del valore di A nella espressione.

Incremento & decremento: esempi

```
int A=10, B;
char C='a';

B=++A;        /*A e B valgono 11 */
B=A++;              /* A=12, B=11 */
C++;                /* C vale 'b' */

int i, j, k;
k = 5;
i = ++k; /*  i = 6, k = 6 */
j = i + k++; /* j=12, i=6,k=7 */

j = i + k++; /*equivale a:j=i+k; k=k+1;*/
```

- In C l'ordine di valutazione degli operandi non e' indicato dallo standard: si possono scrivere espressioni il cui valore e` difficile da predire:

```
k = 5;
j = ++k * k++; /* quale effetto ?*/
```

Operatori di assegnamento abbreviato

E` un modo sintetico per modificare il valore di una variabile.

Sia v una variabile, op un'operatore (ad esempio, +,-,/, etc.), ed e una espressione.

v op = e

equivale a:

v = v op (e)

Ad esempio:

```
k += j;              /*  equivale a k = k + j  */
k *= a + b;          /*  equivale a k = k * (a + b)  */
```


Operatore sequenziale

Un'espressione sequenziale (o di ***concatenazione***) si ottiene concatenando tra loro più espressioni con l'operatore virgola (,).

(<espr1>, <espr2>, <espr3>, .. <esprN>)

- Il risultato prodotto da un'espressione sequenziale e` il risultato ottenuto dall'ultima espressione della sequenza.
- La valutazione dell'espressione avviene valutando nell'ordine testuale le espressioni componenti, **da sinistra verso destra**.

Esempio:

```
int A=1;
char B;
A=(B='k', ++A, A*2);        /* A=4 */
```

Precedenza e Associatività degli Operatori

In ogni espressione, gli operatori sono valutati secondo una **precedenza** stabilita dallo standard, seguendo opportune regole di **associatività**:

- La **precedenza** (o priorità) indica l'ordine con cui vengono valutati operatori diversi;
- L'**associatività** indica l'ordine in cui operatori di pari priorità (cioè, stessa precedenza) vengono valutati.

➔ E` possibile forzare le regole di precedenza mediante l'uso delle parentesi.

Precedenza & associatività degli operatori C

Precedenza	Operatore	Simbolo	Associatività
1 (max)	chiamate a funzione selezioni	`()` `[]` `->` `.`	a sinistra

2	operatori unari:
	op. negazione
	op. aritmetici unari
	op. incr. / decr.
	op. indir. e deref.
	op. sizeof
		
		`!` `~`
		`+` `-`
		`++` `--`
		`&` `*`
		`sizeof`
			a destra

3	op. moltiplicativi	`*` `/` `%`	a sinistra
4	op. additivi	`+` `-`	a sinistra

Precedenza & associatività degli operatori C (continua)

Precedenza	Operatore	Simbolo	Associatività
5	op. di shift	>> <<	a sinistra
6	op. relazionali	< <= > >=	a sinistra
7	op. uguaglianza	== !=	a sinistra
8	op. di AND bit a bit	&	a sinistra
9	op. di XOR bit a bit	^	a sinistra
10	op. di OR bit a bit	|	a sinistra
11	op. di AND logico	&&	a sinistra
12	op. di OR logico	||	a sinistra
13	op. condizionale	? . . . :	a destra
14	op. assegnamento e sue varianti	= += -= *= /= %= &= ^= |= <<= >>=	a destra
15 (min)	op. concatenazione	,	a sinistra

Esempi

Sia V=5, A=17, B=34. Determinare il valore delle seguenti espressioni :

A<=20 || A>=40 → ***vero*** {A<20}
! (B=A*2) → ***falso*** {B=34=17*2}
A<=B && A<=V → ***falso*** {A>V}
A<=(B&&A)<=V → ***vero*** {A<=1<=V}
A>=B && A>=V → ***falso*** {A<B}
! (A<=B && A<=V) → ***vero***
! (A>=B) || !(A<=V) → ***vero***
(A++, B=A, V++, A+B+V++) → ***42*** (A=18, B=18, V=7)