Tipi di dati in C

Tipi di dati

- Un **Tipo di dato** è determinato da un *insieme* di elementi e di *operazioni* definite sugli elementi.
- In C i tipi base sono i **tipi interi** e i **tipi reali**.

Tipi interi

- Permettono la rappresentazione di insiemi di interi.
- I tipi che vedremo sono `int` e `char`.
- Tramite i qualificatori `signed` e `unsigned` si ottengono altri tipi.
- Ecco la classificazione completa:

  char signed char unsigned char
  int unsigned int

Tipi interi

- La differenza principale consiste nella *dimensione* di tali tipi, ossia dal numero di byte utilizzati per memorizzare un elemento del tipo.
- Per conoscere la dimensione di un tipo usare operatore `sizeof`.
- Un tipo intero `T` (`T = char` o `T = int`) di dimensione n permette di rappresentare un insieme di 2^n numeri. Più precisamente:
  - Il tipo `unsigned T` rappresenta l'insieme $\{0, \ldots, 2^n - 1\}$ (numero binario positivo);
  - Il tipo `signed T` rappresenta l'insieme $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$ (numero in complemento a due).

Tipi interi

- La dimensione di un tipo dipende dal sistema usato. L'ANSI C stabilisce che:

$$sizeof(char) = 1$$

$$sizeof(signed\ T) = sizeof(unsigned\ T) = sizeof(T)$$

  dove $\mathbf{T}$ è un qualunque tipo intero.
- I valori massimo e minimo di ciascun tipo sono determinati da costanti definite nel file *limits.h*.

Alcuni valori

```
#include <stdio.h>
#include <limits.h>
int main(void)
{
   printf("int = %d\n",sizeof(int));
   printf("char= %d\n",sizeof(char));
   printf("unsigned char= %d\n",sizeof(unsigned char));
   printf("Valore di SCHAR_MAX : %d\n", SCHAR_MAX);
   printf("Valore di SCHAR_MIN : %d\n", SCHAR_MIN);
   printf("Valore di UCHAR_MAX : %d\n", UCHAR_MAX);
   return 0;
}
```

Tipi interi

Generalmente `char` equivale a `signed char`. Per verificare se questo è vero provare il programma:

```
#include <stdio.h> int main(void)
{
   char c= -1;
   signed char s= -1;
   unsigned char u= -1;
   printf("c = %d s= %d u = %d\n", c, s ,u);
   return 0;
}
```

Tipi interi

- Per capire il funzionamento, si ricordi che nella rappresentazione **complemento a 2** il valore -1 è rappresentato dal byte 11111111; d'altra parte, visto come "intero senza segno", il numero binario 11111111 corrisponde a 255.
- L'output corrispondente alle variabili s e u è: s=-1 u=255
- Se viene stampato `c= -1` significa che `char` è sinonimo di `signed char`, se invece si ottiene `c= 255` significa che `char` sta per `unsigned char`.

Costanti

- L'ANSI C permette di definire costanti di ogni tipo. Esempi di costanti intere sono: 123 <– costante di tipo int
- Un **carattere** fra apici è una **costante intera**. Ad esempio:
  - Il carattere `'a'` rappresenta l'intero 97
  - Il carattere `'A'` rappresenta l'intero 65
  - Il carattere `'0'` rappresenta l'intero 48.
- Quindi il C **non** fa distinzione fra caratteri e interi, ma vengono trattati allo stesso modo e su di essi sono definite le stesse operazioni.

Uso di `printf`

Carattere di conversione	Formato
d	intero in notazione decimale
c	carattere

Esempio:

```
printf(“a%cbe%co”,’l’,’r’ );
printf(“a%cbe%co”, 108, ’r’);
printf(“a%cbe%co”, ’l’ , 114);
printf(“a%cbe%co”, 108 , 114);
```

visualizzano la parola `albero`.

Invece `printf(“a%cbe%do", 108, ’r’);`

visualizza la stringa `albe114o`.

Uso di `printf (3)`

Provare a eseguire il seguente programma che stampa alcune costanti intere usando i due formati visti (osservare l'effetto prodotto dalla stampa dei caratteri **newline** e **bell**).

```
#include <stdio.h> int main(void)
{
   int a, b, c, d, e;
   a=49; b=65; c=97; d=7; e=10;
   printf("%d %d %d %d %d",a,b,c, d,e);
   printf("%c %c %c %c %c",a,b,c, d,e);
   a='1'; b='A'; c='a'; d='‘; e='\n';
   printf("%d %d %d %d %d",a,b,c, d,e);
   printf("%c %c %c %c %c",a,b,c, d,e);
   return 0;
}
```

Operatori aritmetici

- Sui tipi interi sono definiti gli operatori aritmetici + (somma), - (differenza), * (prodotto), / (divisione), % (modulo).
- In C uno dei concetti più importanti è quello di **espressione**.
- La definizione completa di espressione è piuttosto complessa. Di seguito daremo una versione **molto** semplificata.

Operatori aritmetici (2)

Espressione	**Valore**
$const$	Valore della costante aritmetica $const$
var	Contenuto della variabile var
$f(p_1, \ldots, p_n)$	valore restituito dalla chiamata di funzione $f(p_1, \ldots, p_n)$
(E_1 + E_2)	Valore(E_1)+Valore(E_2)
(E_1 - E_2)	Valore(E_1)-Valore(E_2)
(E_1 * E_2)	Valore(E_1) * Valore(E_2)
(E_1/E_2)	Divisione intera fra Valore(E_1) e Valore(E_2)
(E_1 % E_2)	Resto della divisione intera fra Valore(E_1) e Valore(E_2)

Esempio

char a=10;
int b='a';

Espressione	Valore
`a + 87`	97 (o, equivalentemente, 'a')
`a + 'a'`	107 (oppure 'k')
`b + 1`	98 (oppure 'b')
`b + '1'`	146
`'b'+ 1`	99 (oppure 'c')
`'b'+ '0'`	146
`a / 7`	1
`a % 7`	3

I tipi reali

- I tipi reali che considereremo sono
  float
  double
- La differenza consiste nella loro dimensione (numero di bit utilizzati per la mantissa e l'esponente), quindi nel numero di valori rappresentabili.
- Esempi di costanti reali:

```
2.5F /* F (oppure f) denota una costante di tipo float */
1.234 /* costante di tipo double */
```

I tipi reali

- Sui reali sono definiti gli operatori
  `+` (somma),
  `-` (sottrazione),
  `*` (prodotto) e
  `/` (divisione).
- Per denotare una costante di tipo `double` occorre usare il punto. Ad esempio, 3 e 3.0 sono due costanti diverse: la prima è di tipo `int` mentre la seconda è di tipo `double`.

Stampa di numeri reali

- Per stampare dei valori reali con `printf()` occorre usare opportuni caratteri di conversione

Carattere di conversione	**Formato**
`f`	numero in virgola mobile
e	numero in virgola mobile in notazione esponenziale

Stampa di numeri reali: Esempio

```
#include <stdio.h> int main(void)
{
   double a=34.5687897;
   double b=1.23e4;
   double c=12e-5;
   printf("Valore di a: %f\n %e\n", a,a);
   printf("Valore di b: %f\n %e\n", b,b);
   printf("Valore di c: %f\n %e\n", c,c);
   return 0;
}
```

Lettura di numeri reali

- Si usa `scanf()` con opportune specifiche di conversione

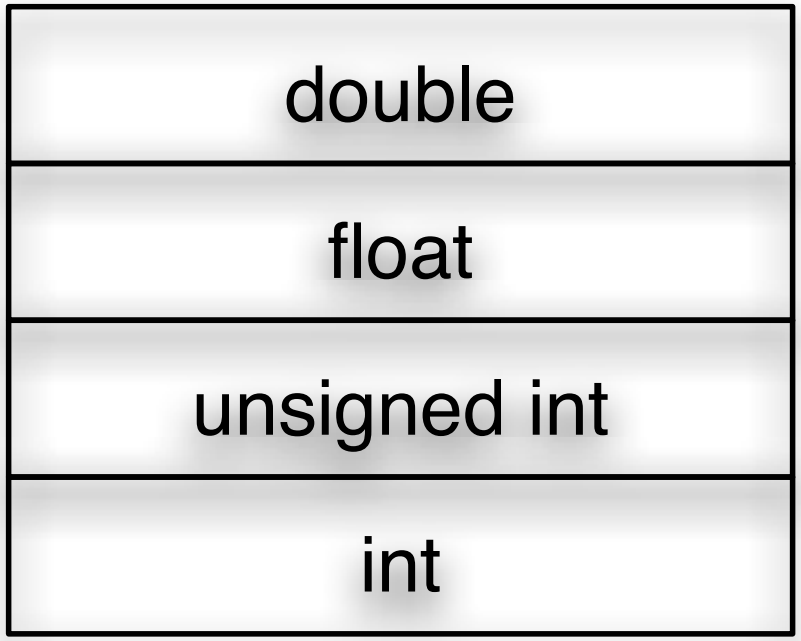
Carattere di conversione	**Formato**
`f`	numero in virgola mobile (di tipo `float`)
`lf`	numero in virgola mobile (di tipo `double`)

Lettura di numeri reali: Esempio

```
#include <stdio.h>
int main(void)
{
    double raggio;
    raggio = 1.0;
    while(raggio != 0.0)
    {
        printf(“Inserire raggio (0 per terminare): “);
        scanf("%lf", &raggio);
        if(raggio= 0.0)
        {
            printf(“Perimetro: %f”, 2*3.14*raggio);
            printf(“\tArea: %f\n”, raggio*raggio*3.14);
        }
    }
}
```

Conversioni di tipo

Quando un operatore binario viene applicato a espressioni i cui valori hanno tipo diverso (espressioni *miste*), questi vengono *automaticamente* convertiti a uno stesso tipo. In genere vengono implicitamente effettuate le conversioni che fanno passare da un tipo a uno “più grande” senza perdita di informazione.



Conversioni di tipo

L'operando "più piccolo" viene automaticamente convertito al tipo "più grande" che risulta poi essere il tipo dell'espressione.

```
char c; int i; float f; double d;
c=20; i=1000; f=1.5; d=-2.5;
```

Espressione	Tipo dell'espressione	Valore
c + 5	`int`	25
c + 'd'	`int`	120
c + 5.1	`double`	25.1
i / 3	`int`	333
i / 3.0	`double`	333.33...
f + 'f'	`float`	103.5

Conversioni di tipo

Le espressioni
```
31 / 2
31 / 2.0
```
hanno significato diverso. Nella prima ’`/`’ è l’operatore di divisione fra interi, quindi il valore dell’espressione è 15. Nella seconda, essendo la costante 2.0 di tipo `double`, ’`/`’ è l’operatore di divisione fra reali; 31 viene convertito a `double` e l’espressione ha tipo `double` e valore 15.5.

Operatori relazionali e di uguaglianza

- Per confrontare il valore di due espressioni (aventi tipo intero o reale) si possono usare gli operatori di confronto < (*minore*), <= (*minore o uguale*), > (*maggiore*), >= (*maggiore o uguale*), == (*uguale*), `!=` (*diverso*).
- Se $\mathcal{R}$ è uno di tali operatori, E_1 e E_2 sono espressioni, allora

$$E_1 \ \mathcal{R} \ E_2$$

  è una espressione di tipo `int` che vale 1 se $E_1 \ \mathcal{R} \ E_2$ è verificata, 0 altrimenti (se E_1 e E_2 hanno tipo diverso vengono effettuate conversioni implicite nel modo prima descritto).
- Si noti come la *condizione booleana* $E_1 \ \mathcal{R} \ E_2$ è vista in C come una *espressione intera*.

Operatori relazionali e di uguaglianza: Esempi

Espressione	**Valore**
`5 < 3`	`0`
`7.23 > (3+1)`	1 (conversione a `double` di $3+1$)
`'a' < 'f'`	1
`'a' + 2 > 'b'`	1
`98.1 - 1.1 == 'a'`	1 (conversione a `double` di 'a')
`(5.5 > -2) + 3`	`4` (tipo `int`)
`(5.5 > -2) + 3.0`	`4.0` (tipo `double`)
`(10 <= 11)== 1`	1
`(10 <= 11)== 0`	0

Priorità e associatività di un operatore

Per ogni operatore è definita la **priorità** e la **associatività**.

- La **priorità** serve per determinare l'ordine con cui gli operatori vanno applicati; vengono eseguiti per prima gli operatori con priorità più alta.
  Ad esempio, poichè `*` ha priorità maggiore di +, l'espressione `3 * 5 + 20 * 4` va interpretata come `(3 * 5) + (20 * 4)`

Priorità e associatività di un operatore

La **associatività** serve per determinare l'ordine con cui vanno applicati gli operatori aventi la stessa priorità. L'associatività può essere *da destra a sinistra* oppure *da sinistra a destra*.

Ad esempio, + e - hanno la stessa priorità e hanno entrambi associa-

tività da sinistra a destra. Quindi, l'espressione `10 - 20 + 1.5` equivale a

`(10 - 20) + 1.5`

La tabella degli operatori

Gli operatori con priorità più alta sono riportati nella prima riga; scendendo verso il basso la priorità diminuisce. Gli operatori sulla stessa riga hanno uguale priorità.

Operatori	Associatività
`sizeof` `(...)` `+` (*unario*) `-` (*unario*) `*` (*deriferimento*) `&` (*indirizzamento*)	⇐
`*` (*prodotto*) `/` (*quoziente*) `%` (*modulo*)	⇒
`+` (*somma*) `-` (*differenza*)	⇒
`<` `<=` `>` `>=` (*operatori di confronto*)	⇒
`==` `!=` (*operatori di confronto*)	⇒

Esempi

Si supponga di aver definito `int i = 10; double d = 1.5;`

Espressione	Espressione equivalente	Valore
`5 * -3 * 2 + 23`	`((5 * (-3)) * 2) + 23`	-7
`30 > 20 > 7`	`(30 > 20) > 7`	0
`d + i == 10`	`(d + i) == 10`	0
`7.1 + d + (i == 10)`	`((7.1 + d) + (i == 10))`	9.6
`i == 10 != 10`	`(i == 10) != 10`	1
`20 == 7 < i + 19`	`20 == (7 < (i + 19))`	0
`20 == (7 < i) + 19`	`20 == ((7 < i) + 19)`	1
`i == i == i`	`(i == i) == i`	0
`i == i == 1`	`(i == i) == 1`	1

typedef

Il C fornisce il meccanismo `typedef`, che permette di associare un tipo a un identificatore. Ad esempio:

```
typedef char caratteri;
typedef int interi;
```

Ogni nome di tipo cosÏ definito può essere utilizzato successivamente per definire variabili o funzioni come se fossero tipi fondamentali.

```
typedef char caratteri;
typedef int interi;
caratteri f(interi p)
{
   interi valore;
   ...
}
```

Attenzione!!! La definizione di un nuovo tipo non riserva spazio in memoria centrale.

Variabili strutturate: Vettori

Un Vettore od Array è un insieme ordinato di una o più variabili tutte dello stesso tipo: <tipo> <nome>[<numero-elementi>]

- `<tipo>` è il tipo degli elementi che costituiscono l'insieme.
- `<nome>` è un identificatore che viene utilizzato per riferirsi all'insieme.
- `<numero-elementi>` è il numero totale di elementi che costituiscono l'insieme.

```
int valori[20];
float elementi[12];
char gradi[12];
```

Vettori

- Un vettore può essere immaginato come un contenitore suddiviso in tanti scomparti od **elementi**.
- Ogni elemento contiene un unico dato del tipo indicato nella definizione.
- Ogni elemento è individuato da un numero progressivo, **indice**, che specifica la posizione dell'elemento all'interno del vettore.
- L'indice può assumere valori interi da 0 al numero totale di elementi meno 1.
- Il numero complessivo di elementi del vettore è la **lunghezza**.

1.3	2.1	7.5	-1.1	2.8	9.2	2.4	11.2	65.0	0.9
0	1	2	3	4	5	6	7	8	9

Vettori

- Il tipo dei dati contenuti nel vettore viene detto **tipo del vettore**.
- Per accedere ad un singolo elemento di un vettore si deve specificare il nome del vettore seguito dall'indice dell'elemento

posto tra parentesi quadre.

```
int valori[20];
int x;
. . .
valori[7] = 5;
x = valori[7];
```

- Ogni elemento di un vettore è una variabile del tipo del vettore.

```
valori[7] /* e' una variabile di tipo int */
```

Vettori

Spesso i vettori vengono trattati all'interno di iterazioni poichè risulta semplice fare riferimento ad i suoi elementi incrementando il valore di una variabile

```
int x[10];
int i;
. . .
for (i=0; i<10; i=i+1)
{
   printf("Inserisci un valore: ");
   scanf("%d", &x[i]);
}
```

Vettori

- Un vettore **non** può essere copiato od assegnato come un unico oggetto.

```
int x[10], y[10];
. . .
x = y; /* Errore */
```

- I vettori **non** possono essere confrontati come unici oggetti.

Vettori e funzioni

- Una funzione può definire un parametro formale di tipo vettore.

```
void ordina (int a[10])
{
   . . .
}
```

- Non è necessario però indicare la dimensione del vettore.

```
void ordina (int a[])
{
   . . .
}
```

Vettori e funzioni

- Quando il nome di un vettore viene passato ad una funzione, quello che viene passato è l'indirizzo dell'elemento iniziale.

```
int x[10];
. . .
ordina (x);
```

- All'interno della funzione chiamata il parametro formale corrispondente al vettore è una variabile contenente un indirizzo.

```
void ordina (int a[])
{
   . . .
}
```

- Accedere in lettura o scrittura ad un parametro di tipo vettore corrisponde ad accedere in lettura o scrittura al corrispondente parametro attuale.

Vettori: Esempio (massimo di un vettore)

```
#include <stdio.h>
int posizioneMax (int [], int);
int main(void)
{
   int x[4], i, max;
   for (i = 0; i < 4, i = i + 1)
   {
      printf("Inserisci un valore: ");
      scanf("%d", &x[i]);
   }
   max = posizioneMax (x, 4);
   printf("Max = %d\t Posizione = %d\n", x[max], max);
   return 0;
}

int posizioneMax (int v[], int t)
{  int i, temp;
   temp = 0;
   for (i = 1; i < t; i = i + 1)
      if (v[i] > v[temp]) temp = i;
   return temp;
}
```

Vettori e `typedef`

- La `typedef` può essere utilizzata per definire un nuovo nome per un tipo di dato vettore. La sintassi è la seguente: typedef <tipo_del_vettore> <nuovo_nome>[<dimensione>] Ad esempio typedef int vett[100];
- viene definito un nuovo nome, `vett`, che è un sinonimo per tipo di dato vettore di `100` elementi di tipo `int`.

Vettori: Esempio (massimo di un vettore)

```
#include <stdio.h>
int posizioneMax (int [], int);
typedef int VETT[4];
void main()
{
   VETT x;
   . . .
}
int posizioneMax (VETT v, int t)
{
   . . .
}
```

Vettori: Esempio (ordinamento di un vettore)

```
#include <stdio.h> typedef int VETT[4];
void ordina (VETT, int);
int posizioneMax (VETT, int);
int main(void)
{
    VETT x;
    int i;
    for (i = 0; i < 4, i = i + 1)
    {
        printf("Inserisci un valore: ");
        scanf("%d", &x[i]);
    }
    ordina(x, 4);
    for (i = 0; i<4, i=i+1)
        printf("%d\t", x[i]);
    return 0;
}
```

Vettori: Esempio

```
void ordina (VETT v, int t)
{
   int i, temp, max;
   for (i = t; i > 1; i = i - 1)
   {
      max = posizioneMax(v, i);
      temp = v[i - 1];
      v[i] = v[max];
      v[max] = temp;
   }
}
```

Variabili strutturate: Strutture

Una Struttura è una collezione di una o più variabili, di uno o più tipi:
struct <tag> { <lista-elementi> };

- La parola chiave `struct` introduce una dichiarazine di struttura.
- `<tag>` è chiamato tag della struttura. É un identificatore opzionale che può essere utilizzato come abbreviazione per la parte di dichiarazione compresa tra le parentesi.
- `<lista-elementi>` è una lista di dichiarazioni di variabili.

```
struct quadrato
{
   int lato;
   float area;
};
```

Strutture

- Le variabili nominate nella struttura sono dette **campi** od **elementi**.
- Un campo od un tag ed una variabile ordinaria (non appartenente ad alcuna struttura) possono avere lo stesso nome.
- I nomi dei campi possono ricorrere in strutture diverse.

```
int lato;
int area;
struct a
{
    float area;
};
struct lato
{
    float area;
};
```

Strutture

- La dichiarazione `struct` definisce un tipo di dato.
- La parentesi graffa di chiusura può essere seguita da una lista di variabili:
  `struct datiCerc { float area; } x, y, z;`
- In un'espressione, un campo di una particolare struttura viene referenziato attraverso un costrutto del tipo:
  nome-struttura.campo

```
struct quadrato
{
    int lato;
    float area;
} x;
. . .
printf("%f", x.area);
. . .
```

Strutture

- L'operatore di campo di una struttura "." connette il nome della struttura con quello del campo.
- Le strutture possono essere innestate l'una nell'altra.

```
struct linea
{
    int dim;
};
struct rettangolo
{
    struct linea altezza;
    struct linea lunghezza;
};
```

Strutture

- Una struttura può essere soltanto copiata, assegnata come un unico oggetto oppure manipolata tramite l'accesso ai suoi campi.

```
struct p { int a; } x, y;
. . .
x = y;
```

- la copia comprende anche il passaggio di argomenti alle funzioni e la restituzione di valori dalle funzioni.
- le strutture possono essere passate alle funzioni per valore e per indirizzo.

```
int perimetro (struct rettangolo a) { . . . }
struct linea altezza (struct rettangolo *a) { . . . }
```

- Le strutture non possono essere confrontate.

Strutture e `typedef`

- La `typedef` può essere utilizzata per definire un nuovo nome per un tipo di dato struttura. La sintassi è la seguente: typedef <definizione_di_struttura> <nuovo_nome>
  Ad esempio typedef struct {int a;} str;
- viene definito un nuovo nome, `str`, che è un sinonimo per tipo di dato `struct {int a;}`.

Strutture: Esempio

```
#include <stdio.h>
#include <math.h>
typedef struct
{
    int area;
    float diagonale;
} quadrato;
void calcolaValori (quadrato *, int);
float visualizzaDiagonale (quadrato);
```

Strutture: Esempio

```
int main(void)
{
    quadrato x;
    int lato;
    float diagonale;
    printf(“Inserisci il lato del quadrato: “);
    scanf(“%d”, &lato);
    calcolaValori (&x, lato);
    printf(“Area: %d”, x.area);
    printf(“Diagonale: %f”, visualizzaDiagonale (x));
    return (0);
}
```

Strutture: Esempio

```
void calcolaValori (quadrato *q, int l)
{
   (*q).area = l * l;
   q->diagonale = sqrt(2) * q->area;
/* (*q). e' equivalente ad q-> */
}
float visualizzaDiagonale (quadrato q)
{
   return q.diagonale;
}
```

La tabella degli operatori trattati

Operatori	Associatività
`->` `.` (*campo di una struttura*)	$\Rightarrow$
`sizeof` `(...)` `+` (*unario*) `-` (*unario*) `!` (*negazone*) `*` (*deriferimento*) `&` (*indirizzamento*)	$\Leftarrow$
`*` (*prodotto*) `/` (*quoziente*) `%` (*modulo*)	$\Rightarrow$
`+` (*somma*) `-` (*differenza*)	$\Rightarrow$
`<` `<=` `>` `>=` (*operatori di confronto*)	$\Rightarrow$
`==` `!=` (*operatori di confronto*)	$\Rightarrow$
`&&` (*AND logico*)	$\Rightarrow$
`||` (*OR logico*)	$\Rightarrow$