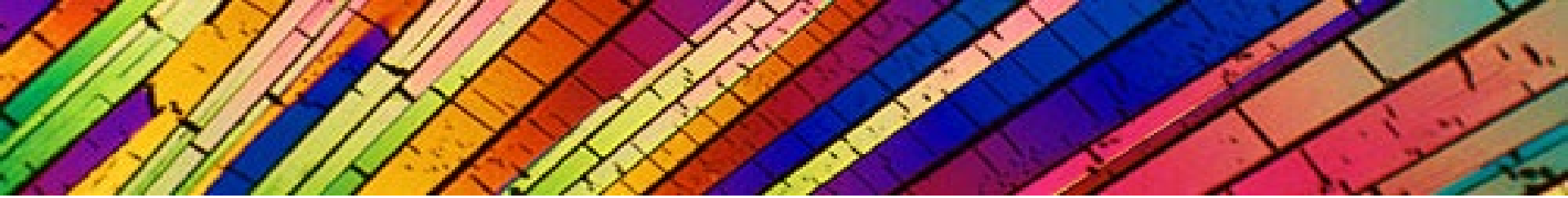

Tipi numerici di dati di base

Ver. 3

Variabili

- Sono porzioni di memoria RAM usate per mantenere dati variabili nel tempo
- La ***definizione*** di una variabile riserva una porzione di memoria (*allocazione*) adeguata a contenere un valore del tipo di dato indicato (intero, floating-point, ecc.)
- Definizione

  tipoDato nomeVar `[,` *nomeVar*`]... ;`
- Esempi

```
int x;
float y, k, t;
```

Variabili

- Le *definizioni* delle variabili sono collocate *tutte all'inizio del blocco* dove sono utilizzate, prima di tutte le istruzioni eseguibili
- Il C99 permette la definizione delle variabili anche tra le istruzioni eseguibili, purché la definizione avvenga prima dell'uso, in C89 non si può fare
- Il termine *dichiarazione* in C ha un altro significato completamente diverso: indica che la definizione è situata altrove (e quindi non alloca memoria), verrà trattato in seguito

Variabili

- Le variabili sono create (allocate) quando si entra nel blocco dove sono definite e vengono eliminate quando si esce dal blocco stesso, si dice che hanno ***allocazione automatica***
- Una variabile non è necessariamente utilizzabile in ogni parte del programma, ma eventualmente solo in alcune parti del codice
- La regione di codice dove una variabile è utilizzabile (visibile) è detta ***ambito di visibilità*** (***scope***, pronuncia: /scoup/)
- Lo scope di una variabile si estende dal punto dove viene definita fino al termine del blocco che la contiene (parentesi graffa di chiusura)

Variabili

- I nomi delle variabili sono ***identificatori*** e seguono le regole già indicate
- La lunghezza massima dei nomi è diversa nel caso di identificatori definiti in altri *file oggetto* o librerie (trattato più avanti, non ci si preoccupi di questo ora)
- *È bene usare nomi di variabili significativi del loro contenuto*
- Le variabili sono convenzionalmente scritte in minuscolo:

```
somma          somma2          somma_2
somma_al_quadrato   sommaAlQuadrato
```

Tipi di dati

- Il tipo di una variabile indica quali valori possono essere memorizzati in essa: intero, floating-point, ecc.
- I bit che costituiscono una variabile assumono significato in base al tipo di dato, ad es. supponendo una sequenza di 32 bit:
  - se si tratta di un intero in complemento a due, l'intera sequenza rappresenta un valore intero con segno
  - se si tratta di un floating-point, la sequenza è suddivisa in segno, esponente e mantissa che insieme rappresentano un valore frazionario in formato esponenziale

Tipi interi di base

- Il C dispone dei seguenti tipi di base:
  - `char` valore intero su un byte
  - `short int` valore intero con segno, in genere ha meno bit di un `int`
  - `int` valore intero con segno, è di solito pari alla *word macchina*
  - `long int` valore intero con segno, in genere ha più bit di un `int`
- Con `short` e `long` di solito si omette `int`

Il tipo char

- Il tipo `char` è un normale tipo intero composto da pochi bit (8), può dunque essere usato in ogni contesto ove basti un numero intero piccolo (compatibile con l'intervallo dei valori rappresentabili con un `char`, vedere più avanti)
- Come il suo stesso nome indica, è il tipo di dato da preferire per contenere il codice ASCII di un carattere

Compatibilità dei tipi interi

- Dimensioni degli interi secondo lo standard:
  - `short`: almeno 16 bit
  - `int`: almeno 16 bit e non meno di uno `short`
  - `long`: almeno 32 bit e non meno di un `int`
- Valori tipici:

  `short`: 16 bit, `int`: 32 bit, `long int`: 64 bit
- Ciascuno dei tipi più capienti può contenere un valore di tutti i tipi meno capienti (es. una variabile `short` può contenere un valore di tipo `char` e una `int` un valore di tipo `short` (e quindi anche `char`), ecc.

Interi senza segno

- Tutti i tipi interi (`char`, `int`, `short`, `long`) possono essere o con segno (tipicamente in Complemento a 2) o senza segno (in binario puro) per avere un range doppio di valori (tutti positivi)
- Per specificare questo, davanti alla definizione della variabile si può aggiungere una delle due keyword: `signed` o `unsigned`
- Esempi

```
unsigned short int z;
signed char x;
unsigned long y;
```

Interi senza segno

- Per i tipi `int`, `short`, `long`:
  il default è `signed` (e quindi non è necessario specificarlo)
- Per il tipo `char`:
  se siano `signed` o `unsigned` dipende dal compilatore in quanto lo standard non lo specifica, questo richiede solo che i caratteri *stampabili* siano sempre memorizzati come valori positivi. Tipicamente sono `signed` ed è inoltre possibile cambiare il default con opportune impostazioni del compilatore

Tipi integrali

- Nella terminologia dello standard C89, i tipi `int` e `char` vengono collettivamente chiamati "integral types"

Tipi interi: limiti del compilatore

- L'header file `<limits.h>` definisce alcune costanti relative al range dei tipi interi (dipendono dal compilatore, tra parentesi i valori tipici o i minimi richiesti dallo standard):
  - `CHAR_BIT` (numero di bit per `char`, in genere 8)
  - `CHAR_MAX` (in genere se signed +127)
  - `CHAR_MIN` (in genere se signed –127)
  - `SHRT_MAX` (almeno +32767)
  - `SHRT_MIN` (almeno –32767)
  - `INT_MAX` (almeno +32767)
  - `INT_MIN` (almeno –32767)
  - `LONG_MAX` (almeno +2147483647)
  - `LONG_MIN` (almeno –2147483647)

Costanti intere decimali

- Sono sequenze di cifre decimali eventualmente precedute da '+' o '–' *che NON iniziano 0* (altrimenti sono considerate numeri ottali)
- In C89, se la costante numerica è rappresentabile con un `int` il suo tipo è `int` (anche se potrebbe bastare un `char` o uno `short`), altrimenti `long` se è sufficiente, altrimenti `unsigned long`, altrimenti errore:
  - `0` → `int`
  - `123456` (se `int` è su 16 bit) → `long`

Costanti intere decimali

- Mediante un suffisso è possibile specificare costanti di tipo:
  - `long` se seguite da `L` o `l`
    es. `123L`
  - `unsigned` se seguite da U o u
    es. `123U` (`unsigned int`)
    `123UL` (`unsigned long int`)
- Non si possono definire costanti `short` né `unsigned short`
- Se c'è una `U` finale, sono considerati solo i tipi `unsigned int` e `unsigned long`
- Se c'è una `L` finale, sono considerati solo i tipi `long` e `unsigned long`

Costanti intere ottali ed hex

- Una costante intera che inizia con `0` è considerata in base 8 e può avere solo le cifre 0-7: `01317` → vale 719_{10}
- Una costante intera preceduta da `0x` o da `0X` è considerata in base 16 e può avere solo le cifre 0-9, A-F, a-f: `0X2CF` → vale 719_{10}
- Sono di tipo `int` se possibile, altrimenti il primo tra: `unsigned int`, `long`, `unsigned long`, altrimenti si ha errore

Costanti intere ottali ed hex

- Se c'è una `U` finale, la valutazione considera solo i tipi `unsigned int` e `unsigned long`
- Se c'è una `L` finale, la valutazione considera solo i tipi `long` e `unsigned long`
- Se ci sono sia una `U` sia una `L` finali, il topo è `unsigned long`

Costanti intere ottali ed hex

- Esempi

```
014ul    → 12     unsigned long
0xbabau  → 47802  unsigned long int
0xblu    → 11     unsigned long int
```

- Attenzione
  Se un numero hex termina con `E` o `e` viene usato in un'operazione di somma/sottrazione, o deve essere messo tra parentesi o tra la E/e e il +/- deve esserci uno spazio per evitare confusione con i num. esponenziali quali 1E+2:

```
int x = 0xE+2;    → errore
int z = 0xE +2;   → OK
int q = (0xE)+2;  → OK
```

Tipi floating-point di base

- `float` valore in singola precisione
- `double` valore in doppia precisione
- `long double` valore in precisione multipla
- Valori tipici:
  - `float`: 32 bit
  - `double`: 64 bit
  - `long double`: in genere 80 o 128 bit
- Lo standard C89 richiede che:
  - il tipo `double` debba avere almeno lo stesso numero di bit del tipo `float`
  - il tipo `long double` debba avere almeno lo stesso numero di bit del tipo `double`

Tipi floating-point: limiti tipici

- `float` (IEEE-P754)
  - dimensioni: 32 bit
  - range (circa): da $\pm 1.4\times10^{-45}$ a $\pm\ 3.4\times10^{+38}$
  - precisione: circa 7 cifre in base 10, es. -123.4567×10^{-2}
  - valori interi consecutivi rappresentabili: $[-2^{24},+2^{24}]$
- `double` (IEEE-P754)
  - dimensioni: 64 bit
  - range (circa): da $\pm 4.9\times10^{-324}$ a $\pm 1.7\times10^{+308}$
  - precisione: circa 16 cifre (in base 10)
    (ad esempio $-1234567.89012345\times10^{+273}$)
  - valori interi consecutivi rappresentabili: $[-2^{53},+2^{53}]$
- Possono rappresentare i valori ± 0, $\pm\infty$, NaN

Compatibilità dei tipi f.p.

- Una variabile `double` può contenere un qualsiasi valore `float`
- Una variabile `long double` può contenere un qualsiasi valore `double`
- Le variabili floating-point possono contenere tutti i valori dei tipi interi, ma possono esserci delle approssimazioni per i numeri grandi: ad es. un `long` a 32 bit pari a 1234567999 equivale ad un `float` a 32 bit approssimato a circa 1.234567e9, ossia 1234567000

Tipi f.p.: limiti del compilatore

- L'header file `<float.h>` definisce alcune costanti relative alle dimensioni dei tipi di dati floating-point (dipendono dal compilatore, tra parentesi i minimi richiesti dallo standard):
  - `FLT_MAX` (massimo numero positivo di tipo `float`, almeno 10^{+37})
  - `FLT_MIN` (minimo positivo, almeno 10^{-37})
  - `FLT_EPSILON` (almeno 10^{-5}, è il minimo valore `float` x tale che $1.0 + x \neq 1.0$ (ossia non produce underflow)
  - `DBL_MAX` (almeno 10^{+37})
  - `DBL_MIN` (almeno 10^{-37})
  - `DBL_EPSILON` (almeno 10^{-9})

Approssimazione nel f.p

- Quando si passa da una base a un'altra (es. da 10 a 2), i valori non interi possono richiedere un numero infinito di cifre dopo la virgola (0.1)
- La necessità di limitarsi a un numero di bit (24, 53) implica un'approssimazione del valore realmente memorizzato
- Ad esempio 0.1_{10} viene memorizzato in `float` come: 0.10000000149…, questo significa che i calcoli utilizzeranno questi valori approssimati, anche se nella visualizzazione (`printf`) gli stessi valori vengono arrotondati e 0.1 apparirà come 0.1 e non il valore memorizzato

Approssimazione nel f.p

- Esempio pratico

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
printf("%f\n", c);    → visualizza 0.300000
printf("%f\n", a+b); → visualizza 0.300000
```

- Ma internamente `a+b` non è uguale a `c` e un confronto darebbe 0 (falso)
- L'approssimazione non è in genere un problema, ma bisogna conoscere il funzionamento per saperlo gestire (sui confronti si veda il set di slide corrispondente)

Costanti floating-point

- Contengono un *punto* (non la virgola) decimale e/o un esponente intero (positivo o negativo)
- L'esponente è preceduto da una `e` o `E` che ha il significato di "*per 10 elevato a*": `4e5` → $4x10^5$
- Sono di tipo `double` salvo indicazione diversa:
  - `12.0  12.  .23  0.0  0.  .0`
    `21.2  12.3e5  12.3e+5  65e-5`
- Sono di tipo `float` se sono seguite da `F` o `f`
  - `21.2F  12.3e5F  43e-4F  43e4F`
- Sono di tipo `long double` se seguite da `L` o `l`
  - `21.2L  12.3e5L  43e-4L  43e4L`
- `12F` (errore! manca il `.` e/o l'esponente con `e`)
- `12L` (è un `long int`, non un `long double`)

Scelta del tipo di dato

- Si decide in termini di economia complessiva del programma, in base alle seguenti considerazioni:
  - I tipi interi sono elaborati **molto** più velocemente dei tipi floating-point
  - Più bit ci sono, più lento è il calcolo (in genere)
  - Più bit ci sono, maggiore è l'occupazione di memoria complessiva (soprattutto nel caso di vettori e matrici)
  - La conversione/promozione di un valore richiede tempo (descritto nella parte sulle espressioni)
  - I valori `char` e `short` sono comunque convertiti in `int`

Scelta del tipo di dato

- Esempio
  Si vogliono memorizzare 2 milioni di valori interi compresi tra 0 e 100. I valori sono compatibili ossia nell'intervallo ("range") di tutti i tipi di dati visti, dai `char` (che arrivano almeno fino a 127) ai `long double`, ad esempio si considerino le due soluzioni seguenti:
  - si usa il tipo `char`: si occupano 2 MB e i calcoli sono veloci
  - si usa il tipo `double`: si occupano 16 MB e i calcoli sono molto più lenti (senza alcun vantaggio rispetto al caso precedente)

Inizializzazione delle variabili

- La definizione di una variabile *automatica* non le assegna un valore iniziale (il contenuto è indefinito)
- È possibile dare un valore iniziale alle variabili contestualmente alla loro definizione (***inizializzazione***)
- I valori usati per inizializzare possono essere espressioni con costanti e variabili già inizializzate (espressioni trattate più avanti)
- Esempi

```
int x = 12, y = 27*3*x;
double pigreco = 4.0*atan(1.0);
```

Inizializzazione delle variabili

- L'inizializzazione di una variabile ha lo stesso effetto pratico di definizione+assegnazione:

```
int x = 12;
```

equivale a tutti gli effetti a:

```
int x;
x = 12;
```

- Quando possibile, è preferibile assegnare il valore alla variabile non come inizializzazione, ma come assegnazione *poco prima della parte di codice dove viene utilizzata* (per non separare eccessivamente dove viene assegnato il valore e dove viene utilizzato)

Costanti numeriche con nome

- Il modificatore `const` nella definizione delle variabili richiede al compilatore di verificare che queste non vengano mai modificate
- Il valore della variabile `const` deve essere specificato nell'inizializzazione, può anche essere un'espressione di cui viene calcolato il risultato al *run-time* (ossia in esecuzione)

```
const double Pi = 3.141592653;
const double Pi = 4.0*atan(1.0);
```

- `const` viene preferibilmente messo prima del nome di tipo a cui si riferisce, ma le seguenti definizioni sono equivalenti:

```
const int x = 12;
int const x = 12;
```

Enumerazioni

- Un'enumerazione serve per definire una lista di identificatori aventi valori *costanti* di tipo `int`
- Sintassi:
  `enum` *nome* `{`*cost1*`,` *cost2*`, ...};`
- `enum boolean {FALSE, TRUE};`
  definisce le due costanti:
  - `FALSE` (con valore 0)
  - `TRUE` (con valore 1)
- I nomi delle costanti (*enumeration constants*) vengono in genere scritti in maiuscolo
- Al primo identificatore viene assegnato il valore 0, al secondo identificatore 1, ecc.

Enumerazioni

- Il nome dell'enumerazione (detto ***tag***) appartiene al namespace dei tag e in genere viene omesso, a meno che non si vogliano definire successivamente variabili di quel tipo

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

- Ciascun valore può essere inizializzato, anche non in ordine crescente, o no: i successivi proseguono la numerazione

```
enum mesi {GEN=1, FEB, MAR, …};
enum lettere {A=9, B, C=1, D, …};
```

quindi qui `B` vale 10 e `D` vale 2

Enumerazioni

- I valori delle costanti possono ripetersi, anche nella stessa numerazione

```
enum a {A=1, B=1, C=2, D=2, …};
```

- Si possono definire *variabili* di un tipo enumerativo, ma sono in realtà di tipo `int` e il compilatore non è tenuto a verificare se i valori assegnati alla variabile sono tra quelli definiti nella definizione della `enum`:

```
enum boolean {FALSE, TRUE} pippo;
pippo = 12;
```

→*non dà errore anche se* `boolean` *contiene le sole costanti 0 e 1*

Enumerazioni

- I nomi delle costanti devono essere diversi da ogni altro identificatore presente nel modulo dove sono definiti (variabili, funzioni, ecc.)
- È un errore scrivere:

```
enum unodue {UNO, DUE};
enum unotre {UNO, TRE};
```

- Altro esempio di errore:

```
enum costanti {ALFA, BETA, GAMMA};
int ALFA;
```

- Le costanti enumerate sono soggette alle regole di scope

Costanti simboliche

- È possibile dare un nome (un identificatore) a una successione di caratteri, questo nome è detto *simbolo* (per convenzione in maiuscolo)
- Prima della compilazione, il *preprocessore* cerca i simboli definiti con direttive:
  `#define` *nome sequenza_di_caratteri*
  e sostituisce ogni occorrenza del simbolo *nome* con la corrispondente *sequenza_di_caratteri*
- Esempi

```
#define TRUE 1
#define FALSE 0
#define DIM 80
#define INPUT_FILE "miofile.txt"
```

Costanti simboliche

- Nell'esempio precedente, il preprocessore cerca ogni occorrenza di `DIM` e la sostituisce con `80` (i due caratteri `8` e `0`), quindi una riga di codice come la seguente:
  `int vett[DIM];`
  viene trasformata in:
  `int vett[80];`
  prima della compilazione vera e propria
- Si noti che identificatori come `DIMENSIONE` e `ADIMO` non vengono modificati
- Le occorrenze trovate tra doppie virgolette non vengono modificate `"LA DIM DEL TEOREMA"`

Costanti simboliche

- La sostituzione dei simboli inizia a partire dalla riga dove è presente la `#define` e continua fino a fine file (ignorando la struttura a blocchi del programma, ossia non segue le regole di scope come le costanti `enum`)
- Non è un'istruzione C di assegnazione, ma una direttiva del preprocessore, quindi:
  - non bisogna mettere il carattere '=' tra il *nome* del simbolo e la *sequenza_di_caratteri*
  - non si deve mettere il ';' al fondo
- La *sequenza_di_caratteri* può contenere spazi e termina comunque a fine riga

Confronto const, enum, define

- I valori `enum` e `const` vengono allocati esattamente come le variabili, come tali riservano memoria e sono utilizzati dal debugger
- Le costanti simboliche delle `define` sono sostituite dal preprocessore, quindi il compilatore non le vede come variabili, ma come semplici numeri costanti (no debug)
- I valori delle `enum` sono *solo* di tipo `int`
- I valori delle `const` e delle `define` possono essere di qualsiasi tipo

Confronto const, enum, define

- I valori delle `enum` sono automaticamente inizializzati (utile quando si hanno molti valori)
- I valori delle `const` e delle `define` devono essere indicati singolarmente
- La visibilità dei valori delle `enum` e delle `const` è confinata alla funzione dove sono definiti (come semplici variabili)
- La visibilità dei valori delle `define` inizia a partire dalla riga dove è presente la `#define` stessa e continua fino a fine file senza essere confinata dalla struttura a blocchi del C