Variabili, tipi, espressioni in C

Emilio Di Giacomo

Richiami generali

- tipo di dato: specifico dominio di valori

- variabile: "contenitore" che può memorizzare valori di uno specifico tipo di dato, detto tipo della variabile

- espressione: costrutto del linguaggio al quale rimane associato un valore di uno specifico tipo di dato, detto tipo dell'espressione

Tipi in C

- Tipi primitivi:
  - *int*
  - *char*
  - *float*
  - *double*
  - ...
- Tipi strutturati:
  - *array*
  - *struct*
  - ...

Tipi primitivi

- Il C ha quattro tipi primitivi di base:
  - *char*
  - *int*
  - *float*
  - *double*
- e "varianti" che possono essere ottenute utilizzando i modificatori
  - *signed*/*unsigned*
  - *short*/*long*

Tipi primitivi

- L'effettiva implementazione dei diversi tipi non è specificata nello standard
- Pertanto compilatori diversi possono utilizzare implementazioni diverse
- Di conseguenza i valori rappresentabili con ciascun tipo variano da compilatore a compilatore

Tipi interi

- I tipi interi disponibili in C si esprimono combinando i modificatori *long*/*short* e *signed*/*unsigned* con il tipo base *int*
- Esistono combinazioni diverse che corrispondono allo stesso tipo

Tipi interi

	signed	unsigned
	int signed signed int	**unsigned** unsigned int
short	**short** short int signed short signed short int	**unsigned short** unsigned short int
long	**long** long int signed long signed long int	**unsigned long** unsigned long int
long long	**long long** long long int signed long long signed long long int	**unsigned long long** unsigned long long int

In grassetto i "nomi" che useremo da adesso in poi

Tipi interi: signed/unsigned

- I tipi *signed* rappresentano interi con segno
  - permettono cioè di rappresentare sia valori positivi che negativi
  - tipicamente in complemento a 2
- I tipi *unsigned* rappresentano interi senza segno
  - permettono cioè si rappresentare solo valori positivi

Tipi interi: dimensioni

- Lo standard C non specifica il numero di bit da utilizzare per ciascuno dei tipi
- Impone però dei requisiti minimi e dei vincoli sulle dimensioni

Tipi interi: dimensioni

- Vincoli sulle dimensioni:
  - *sizeof(short)* $\leq$ *sizeof(int)* $\leq$ *sizeof(long)* $\leq$ *sizeof(long long)*
  - *sizeof(short)* ≥ 2
  - *sizeof(int)* ≥ 2
  - *sizeof(long)* ≥ 4
  - *sizeof(long long)* ≥ 8
- *sizeof* è una funzione C per conoscere il numero di byte utilizzati per rappresentare un tipo

Tipi interi: dimensioni

- Il file *limits.h* contiene costanti che memorizzano alcune proprietà dei tipi interi, ad esempio:
  - *SHRT_MIN*, *INT_MIN*, *LONG_MIN*, *LLONG_MIN*
    - valori minimi per i tipi signed
  - *SHRT_MAX*, *INT_MAX*, *LONG_MAX*, *LLONG_MAX*
    - valori massimi per i tipi signed
  - *USHRT_MAX*, *UINT_MAX*, *ULONG_MAX*, *ULLONG_MAX*
    - valori massimi per i tipi unsigned

Specificatori di conversione

- Per l'input e l'output (*printf*, *scanf*,...) si usano i seguenti specificatori di conversione per i tipi interi

Tipo	Specificatore di conversione
int	%d o %i
short	%hd o %hi
long	%ld o %li
long long	%lld o %lli
unsigned	%u
unsigned short	%hu
unsigned long	%lu
unsigned long long	%llu

Specificatori di conversione

- Per i tipi unsigned *%u* può essere rimpiazzato da:
  - *%o* per numeri rappresentati in ottale
  - *%x* per numeri rappresentati in esadecimale (utilizzando a, b, c, d, e, f per le cifre maggiori di 9)
  - *%X* per numeri rappresentati in esadecimale (utilizzando A, B, C, D, E, F per le cifre maggiori di 9)

Esercizio

- Scrivere un programma C che verifica il numero di byte utilizzato da ciascuno dei tipi interi e i rispettivi valori massimo e minimo

Espressioni intere e letterali

- I letterali interi e le variabili intere sono espressioni intere elementari
- Con il termine letterale si intende un valore (costante) di un certo tipo rappresentato secondo una certa notazione
  - ogni numero intero (in base 10) è un letterale *int*
  - un intero seguito da una *l* o *L* è un letterale *long*
  - un intero seguito da due *l* o *L* è un letterale *long long*
  - un intero seguito da una *u* o *U* è un letterale *unsigned*
  - i prefissi 0 o 0x permettono di rappresentare il numero in base 8 o 16 rispettivamente

Esempi di letterali interi

- *15* espressione *int* che vale15
- *15L* espressione *long* che vale 15
- *15LL* espressione *long long* che vale 15
- *15u* espressione *unsigned* che vale 15
- *15UL* espressione *unsigned long* che vale 15
- *017* espressione *int* (in ottale) che vale 15
- *0xF* espressione *int* (in esadecimale) che vale 15

Espressioni intere

- Espressioni complesse possono essere create utilizzando gli operatori aritmetici
- Gli operatori aritmetici di base sono:
  - `+` somma
  - `-` sottrazione
  - `*` moltiplicazione
  - `/` divisione intera (tronca la parte decimale)
  - `%` resto della divisione intera

Regole e precedenze

- Gli operatori aritmetici +, -, * funzionano con le classiche regole dell'algebra
- L'espressione *x / y* vale il quoziente della divisione tra *x* ed *y* (es. *10/3* vale *3*)
- Gli operatori *, / hanno precedenza rispetto a +,-
- Gli operatori + e - si possono anche usare come operatori unari (in particolare l'espressione *-a* inverte il segno di *a*)

L'operatore resto

- *x % y* vale il resto della divisione intera tra *x* e *y*
  - ad esempio *10%3* vale *1*
- Più formalmente, deve valere l'equazione:
  *x % y = x – (x/y) * y*
- Quindi ad esempio:

  11 % -5 vale *1*

  -11 % 5 vale *-1*

  -11 % -5 vale *-1*
- L'operatore *%* ha la stessa precedenza di * e /

Parentesi

- Si possono usare parentesi per forzare le regole di precedenza degli operatori
  - si possono usare solo parentesi tonde, con un livello qualunque di annidamento

- Ad esempio, la seguente espressione vale *10*

*((5-2)*7)%3+10/(4-3)*

Ancora sul tipo delle espressioni intere

- Ogni espressione intera che fa uso di operatori viene considerata come espressione:
  - di tipo *int*, se non contiene operandi *long* o *long long*
  - di tipo *long* (o *long long*), se contiene operandi *long* (o *long long*)
    - ad esempio *5 + 5L* è una espressione di tipo *long*

- Osservazione: Se *a* e *b* sono variabili *short*, l'espressione *a+b* è di tipo *int* !!

Assegnazioni di espressioni intere

- Per assegnare il valore di una espressione intera *<esp>* ad una variabile intera *var* dello stesso tipo, basta scrivere

  var = <esp>

- In alcuni casi è anche possibile assegnare espressioni di un certo tipo intero a variabili intere di tipo diverso – analizzeremo più avanti questi casi

Un caso importante

- Sia *a* una variabile intera; considera il seguente frammento di codice:

  a = 12;

  a = a + 5;

- Quanto vale *a* una volta eseguito questo codice?

Un caso importante

- Sia *a* una variabile intera; considera il seguente frammento di codice:

  a = 12;

  a = a + 5;

- Quanto vale *a* una volta eseguito questo codice?
  - Per eseguire l'istruzione *a = a + 5* si valuta prima il valore dell'espressione a destra di = (in questo caso il valore è *17*)
  - Il valore dell'espressione è dunque assegnato ad *a*, sovrascrivendo il precedente valore

Operatori di assegnazione composta

- Istruzioni che incrementano e decrementano il valore di una variabile intera sono molto frequenti
- Per semplificare queste istruzioni, si possono usare gli operatori di assegnazione composta

ESPRESSIONE	ESPRESSIONE EQUIVALENTE
`a = a+b`	`a += b`
`a = a-b`	`a -= b`
`a = a*b`	`a *= b`
`a = a/b`	`a /= b`
`a = a%b`	`a %= b`

Operatori di incremento e decremento

- In particolare per incrementi e decrementi di una unità, si possono usare i così detti operatori di incremento e decremento

ESPRESSIONE	ESP. EQUIVALENTE	ESP. EQUIVALENTE	ESP. EQUIVALENTE
`a = a+1`	`a += 1`	`a++`	`++a`
`a = a-1`	`a -= 1`	`a--`	`--a`

forma postfissa

forma prefissa

- ++ e -- hanno precedenza rispetto agli altri operatori aritmetici

Operatori di incremento e decremento

- L'esecuzione delle istruzioni *a++* e *++a* produce lo stesso effetto sul valore di *a*; ma le due forme possono produrre effetti diversi se usate in espressioni più ampie:

```
int a = 3;
int b = 2 + a++
```

al termine *a* vale *4*
e *b* vale *5*

```
int a = 3;
int b = 2 + ++a
```

al termine *a* vale *4*
e *b* vale *6*

Overflow

- l'overflow è una condizione che si verifica quando assegno ad una variabile intera una espressione dello stesso tipo il cui valore eccede la capacità della variabile stessa
- Ad esempio, assumiamo che il tipo *int* sia rappresentato con 4 byte
- Cosa accade se assegno ad una variabile *int* l'espressione *2147483647 + 1*? (*2147483647* è il massimo valore rappresentabile con 4 byte)
  - non vengono segnalati errori, ma viene assegnato alla variabile un valore errato (nel caso specifico il valore -2^{31} cioè *-2147483648*)

Overflow: motivo

- Sommando *1* alla rappresentazione binaria di *2147483647* si ottiene un riporto, che si propaga fino alla cifra più significativa

$2^{31}-1$: 01111111 11111111 11111111 11111111 +
1: 00000000 00000000 00000000 00000001

-2^{31}: 10000000 00000000 00000000 00000000

Tipi in virgola mobile

- I tipi *float* e *double* e *long double* rappresentano numeri reali secondo la rappresentazione in virgole mobile
- Lo standard non specifica il tipo di rappresentazione che deve essere utilizzata
- L'unico vincolo è:
  - *sizeof(float)* $\leq$ *sizeof(double)* $\leq$ *sizeof(long double)*

Tipi in virgola mobile

- Tipicamente si utilizza:
  - per il tipo *float* la rappresentazione in precisione singola dello standard IEEE 754
  - per il tipo *double* la rappresentazione in precisione doppia dello standard IEEE 754

Letterali in virgola mobile

- I letterali *double* sono costanti numeriche con virgola, espresse usando il "." come separatore tra cifre intere e decimali; si può anche usare la notazione scientifica *xEy*, equivalente a $x10^y$
- Esempio di letterali *double* equivalenti:

  1.23 *0.00123E3* *12.3E-1*

- Si può anche esplicitare un letterale *double*, mettendo il suffisso *d* o *D* al termine del numero

Letterali in virgola mobile

- Una costante numerica con virgola sarà considerata letterale *float* solo se specificato con il suffisso *f* o *F* (esempio *1.23F*)
- Una costante numerica con virgola sarà considerata letterale *long double* solo se specificato con il suffisso *l* o *L* (esempio *1.23L*)

Osservazione sui letterali numerici

- Uno stesso valore costante può essere rappresentato con diversi letterali numerici:
  - se scrivo *53* indico un letterale *int*
  - se scrivo 53.0 oppure 53D indico un letterale *double*
  - se scrivo *53F* indico un letterale *float*

Tipi virgola mobile: dimensioni

- Il file *float.h* contiene costanti che memorizzano alcune proprietà dei tipi in virgola mobile, ad esempio:
  - *FLT_MIN*, *DBL_MIN*, *LDBL_MIN*
    - valori minimi normalizzati per i tipi in virgola mobile
  - *FLT_TRUE_MIN*, *DBL_TRUE_MIN*, *LDBL_TRUE_MIN*
    - valori minimi per i tipi in virgola mobile
  - *FLT_MAX*, *DBL_MAX*, *LDBL_MAX*
    - valori massimi per i tipi in virgola mobile

Specificatori di conversione

- Per i tipi in virgola mobile si usano i seguenti specificatori di conversione
  - per il tipo *float* si usa *%f*
  - per il tipo *double*:
    - con la *scanf* si usa *%lf* (o *%lF*)
    - con la *printf* si usano *%f*, *%g*, *%e* (o *%F*, *%G*, o *%E*)
  - per il tipo long double si usa *%Lf*, *%Lg*, *%Le* (o *%LF*, *%LG*, o *%LE*)

Specificatori di conversione

- Per il tipo *double*:
  - con la *printf* si usano *%f*, *%g*, *%e* (o *%F*, *%G*, o *%E*)
- *%f* visualizza il numero nel formato con la virgola
- *%e* visualizza il numero secondo la notazione scientifica
- *%g* utilizza il formato *%f* o il formato *%e* a seconda del valore
- Un comportamento analogo si ha per il tipo long double con gli specificatori *%Lf*, *%Lg*, *%Le*

Specificatori di conversione

- Esempio:

```
double a=1.23;
double b=1.76E7;
double c=12345667838.0;
printf("%f, %f, %f\n",a,b,c);
printf("%g, %g, %g\n",a,b,c);
printf("%e, %e, %e\n",a,b,c);
```

output:

```
1.230000, 17600000.000000, 12345667838.000000
1.23, 1.76e+007, 1.23457e+010
1.230000e+000, 1.760000e+007, 1.234567e+010
```

Operatori per espressioni con virgola

- Valgono tutti gli operatori usati per le espressioni intere, tranne l'operatore di resto

- L’effetto dell’operatore dipenderà dagli operandi:
  - *15/2* è una espressione intera che vale *7*
  - *15.0/2.0* è una espressione con virgola che vale *7.5*

Espressioni numeriche miste

- E' possibile definire espressioni con operandi numerici di tipo misto, cioè alcuni interi e altri con virgola; il tipo di una espressione mista è quello con dominio più ampio tra quelli coinvolti:
  - *7/2.0* è una espressione *double* di valore *3.5*
  - *7/2 * 3.0* è una espressione *double* di valore *9.0;* infatti, viene prima valutata la sottoespressione *7/2*, che è di tipo *int* e vale *3*; poi viene valutata *3 * 3.0*, che è di tipo *double* e vale *9.0*.

Tipi carattere

- Il tipo utilizzato in C per memorizzare caratteri è il tipo *char*
- Il tipo *char* deve utilizzare almeno un byte
- I caratteri sono rappresentati per mezzo di un codice numerico
  - La codifica utilizzata dipende dal compilatore ma tipicamente è la codifica ASCII estesa (8 bit)

Letterali carattere

- Un letterale carattere rappresenta un singolo carattere è può essere rappresentato da:
  - singoli caratteri racchiusi tra apici
    - es: *'A'*, *'!'*, *'0'*, *'@'*,...
  - sequenze di escape tra apici
    - es: *'\n'*, *'\t'*, *'\\'*, ...
  - sequenze della forma *'\ooo'* dove *ooo* è il codice ottale (da 1 a 3 cifre) del carattere da rappresentare
    - es: *'\101'* è equivalente a *'A'*
  - sequenze della forma *'\xhhh'* dove *hhh* è il codice esadecimale (con un numero qualsiasi di cifre) del carattere da rappresentare
    - es: *'\x41'* è equivalente a *'A'*

Tipo char: dimensioni

- Le seguenti costanti sono definite in *limits.h:*
  - *CHAR_MIN*, *CHAR_MAX*
    - valore numerico minimo e massimo per il tipo *char*
  - *CHAR_BIT*
    - numero di bit utilizzati per il tipo *char*

Specificatore di conversione

- Lo specificatore di conversione per il tipo *char* è *%c*
- Poiché i valori del tipo *char* sono codici numerici è anche possibile visualizzare un *char* come un intero usando lo specificatore *%hhi*
- Ad esempio, il seguente codice

```
char c='A'
printf("%c %hhi", c, c)
```

stampa

```
A 65
```

Espressioni char

- E' possibile formare espressioni *char* combinando variabili e letterali con gli operatori aritmetici
  - l'operatore agisce sui codici numeri dei caratteri coinvolti
  - ad esempio il frammento seguente stampa a video il carattere che occupa la posizione 97; infatti, '0' and '1' occupano le posizioni 48 e 49, rispettivamente

```
char c = '0' + '1';
printf("%c",c);
```

Caratteri e interi

- In realtà il tipo *char* è a tutti gli effetti un tipo intero
- Ad esempio, il seguente codice

```
char a=23;
char b=34;
char c=a+b;
printf("Somma: %hhi",c);
```

stampa

```
Somma: 57;
```

Caratteri e interi

- Poiché *char* è a tutti gli effetti un tipo intero, anche per esso esistono la versione *signed* e la versione *unsigned*
- Il tipo *char* coincide con *unsigned char* o con *signed char* a seconda dei compilatori

Caratteri e interi

- È possibile anche utilizzare tutti gli altri tipi interi per rappresentare i caratteri
- Ad esempio:

```
int a='A';
int b='A'+1;
printf("Carattere: %c",b)
```

stampa

```
Carattere: B
```

Conversioni tra tipi di dati

- E' possibile assegnare ad una variabile di tipo t_1 un valore di tipo t_2?
- In C tale operazione è consentita
- Avviene una così detta conversione implicita (o cast implicito)
  - dal valore dell'espressione di tipo t_2 viene ricavato un valore rappresentato secondo la codifica del tipo di t_1, che viene poi assegnato alla variabile

Perdita di precisione

- Nei casi in cui il dominio di valori di t_2 è contenuto in quello di t_1 la conversione non provoca nessun problema
- Ad esempio il seguente codice non provoca nessun problema:

```
short a = 255;
int b = a;
```

- Nei casi in cui il dominio di valori di t_2 NON è contenuto in quello di t_1 la conversione può provocare una perdita di precisione
- Ad esempio, il seguente codice assegna a b il valore *-31072* (con un compilatore che usa 4 byte per *int* e 2 per *short*)

```
int a = 100000;
short b = a;
```

- i bit in eccesso vengono troncati e si ha una perdita di precisione

Conversione esplicita

- È possibile convertire esplicitamente un valore di un certo tipo ad un valore di un tipo diverso tramite l'operatore di cast
- L'operatore va posto davanti al valore da convertire, ed ha la forma *(t)*, dove *t* è il tipo verso il quale si vuole effettuare la conversione
- Esempio:

```
int a = 9;
int b = 2
float c = (float)a/2;
```

- In questo caso *a* viene convertito a *float* prima di effettuare la divisione e quindi il risultato sarà *4.5*
- Senza il cast il risultato sarebbe stato *4* (divisione fra interi)
- L'operatore di cast ha precedenza rispetto a quelli visti fino qui

Espressioni booleane

- Molto spesso durante l'esecuzione di un programma si ha la necessità di valutare se una certa condizione è vera o falsa
- Espressioni il cui risultato è un valore di verità (vero o falso) si chiamano espressioni booleane in onore del matematico britannico George Boole (1815 - 1864), ideatore dell'algebra che porta il suo nome (algebra booleana):
  - le espressioni booleane vengono anche chiamate predicati

Espressioni booleane

- Molti linguaggi di programmazione (tra cui Java) prevedono esplicitamente un tipo booleano per rappresentare valori di verità
- Il C invece non ha un tipo booleano esplicito
- i valori di verità sono rappresentati da valori interi:
  - il valore *0* corrisponde al valore *falso*
  - qualunque valore diverso da *0* corrisponde al valore *vero*

Espressioni booleane

- E' possibile costruire predicati attraverso l'uso di due tipi di operatori:
  - operatori relazionali: permettono di effettuare confronti tra valori numerici
  - operatori logici: combinano espressioni booleane in espressioni booleane più complesse
- *Nota: il risultato della valutazione di un predicato è concettualmente un valore di verità; in C però tale risultato è un valore intero*

Operatori relazionali

- Gli operatori relazionali C sono i seguenti:
  - == uguale a
  - *!*= diverso da
  - > maggiore di
  - < minore di
  - >= maggiore o uguale a
  - <= minore o uguale a

- Sono tutti operatori binari che hanno lo stesso valore che gli si attribuisce nella matematica

Operatori relazionali: esempi

- Ecco alcuni esempi di semplici espressioni che usano operatori relazionali

Espressione	Risultato	Valore dell' espressione
10>5	Vero	1
10.0<5.0	Falso	0
10.0==10	Vero	1
5<=6	Vero	1
7.4!=7.41	Vero	1
'a'>'z'	Falso	0
'a'>'Z'	Vero	1
'a'>'5'	Vero	1

Attenzione a == e = !!!

- Un errore piuttosto comune consiste nel confondere l'operatore di uguaglianza (==) con l'operatore di assegnazione (=)
- *x==5* è un predicato che verifica se *x* è uguale a *5*
- *x=5* è invece un'operazione di assegnazione che assegna ad *x* il valore *5*
- Nota: poiché in C i predicati sono in realtà espressioni intere, la sostituzione di un == con un = non viene segnalato dal compilatore

Operatori logici

- Gli operatori logici in C sono i seguenti:
  - *&&* AND logico
  - || OR logico
  - *!* NOT (negazione)
- I primi due sono operatori binari, mentre il NOT è un operatore unario

Operatori logici: semantica

- Ecco la tavola di verità degli operatori logici binari

a	b	a && b	a || b
Falso	Falso	Falso	Falso
Falso	Vero	Falso	Vero
Vero	Falso	Falso	Vero
Vero	Vero	Vero	Vero

- Ecco la tavola di verità del NOT

a	!a
Falso	Vero
Vero	Falso

Operatori logici: precedenze

- L'operatore NOT ha precedenza rispetto agli altri e l'operatore AND ha precedenza rispetto all'OR
- E' possibile usare parentesi per forzare le regole di precedenze (come nelle espressioni numeriche)
- Nelle espressioni *a && b* oppure *a || b*, il secondo operando viene valutato solo se necessario
  - cioè a meno che il valore del primo non sia già sufficiente a dedurre il risultato dell'espressione

Operatori logici: esempio

```
#include <stdio.h>

int main(){
    int a;
    printf("Inserire 0 o 1 per a\n");
    scanf("%d",&a);
    int b;
    printf("Inserire 0 o 1 per b\n");
    scanf("%d",&b);
    int c = (a || !b) && !a;
    printf("(a || !b) && !a = %d\n",c);
}
```

In quali casi verrà stampato 1?

Leggi di De Morgan

- Le leggi di De Morgan (o teoremi di De Morgan) permettono di esprimere una espressione con AND sotto forma di una espressione con OR, e viceversa

  - *!(a && b) ⇔ (!a || !b)*
  - *!(a || b) ⇔ !a && !b*

- Quindi, in altra forma:

  - *(a && b) ⇔ !(!a || !b)*
  - *(a || b) ⇔ !(!a && !b)*

Predicati complessi

- E' possibile costruire predicati complessi, combinando operatori logici e relazionali (i secondi hanno precedenza sui primi, ma si possono usare le parentesi per forzare le precedenze)

  - Esempio: se *num* è una variabile numerica, la condizione che *num* cada nell'intervallo *[-7, 7]* oppure nell'intervallo *(10, 14)*, si può scrivere come:
    (num >= -7) && (num <= 7) || *(num > 10) && (num <14)*

  - Esempio: se *num* è una variabile numerica, la condizione che *num* sia pari e diverso da zero si può scrivere come:
    num % 2 == 0 && num != 0

Costanti

- Oltre alla variabili, in C si possono definire dei contenitori chiamati costanti
- Una costante è come una variabile alla quale però può essere dato *un solo valore durante tutto il suo ciclo di vita*
  - il valore di una costante è tipicamente assegnato contestualmente alla sua dichiarazione
- Una costante viene dichiarata come una variabile (tipo + nome), ma la dichiarazione è preceduta dalla parola chiave *const*

Costanti: un esempio

```
#include <stdio.h>

int main(){
    const double PIGRECO = 3.14159; // costante

    printf("Inserire il raggio\n");
    double r;
    scanf("%lf",&r);

    double area = r*r*PIGRECO;
    double perimetro = 2*r*PIGRECO;

    printf("Area del cerchio = %g\n", area);
    printf("Perimetro del cerchio = %g\n", perimetro);

}
```

L’importanza di usare costanti

- Ovviamente è possibile fare a meno delle costanti, usando direttamente i letterali nel codice
  - nell’esempio precedente potevamo scrivere direttamente *3.14159* ovunque usavamo *PIGRECO*

- Tuttavia, usare le costanti consente di modificare in modo minimale il codice in caso di modifica dei valori costanti usati nel codice
  - usare direttamente i letterali può comportare modifiche multiple nel codice

Costanti: definizione alternativa

- In C è possibile definire costanti anche utilizzando la direttiva per il preprocessore *#define*

```
#include <stdio.h>
#define PIGRECO 3.14159
int main(){
    printf("Inserire il raggio\n");
    double r;
    scanf("%lf",&r);

    double area = r*r*PIGRECO;
    double perimetro = 2*r*PIGRECO;

    printf("Area del cerchio = %g\n", area);
    printf("Perimetro del cerchio = %g\n", perimetro);
}
```

Costanti: definizione alternativa

- L'istruzione

```
#define PIGRECO 3.14159
```

definisce una costante simbolica *PIGRECO* il cui valore è *3.14159*

- In fase di preprocessamento tutte le occorrenze della costante simbolica *PIGRECO* saranno rimpiazzate dalla stringa *3.14159*

Costanti: definizione alternativa

- La direttiva per il preprocessore non ha un punto e virgola (;) alla fine
- Se scriviamo

```
#define PIGRECO 3.14159;
```

la variabile simbolica *PIGRECO* verrà rimpiazzata dalla stringa *3.14159;*