



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

IDENTIFICATORI, DICHIARAZIONI, TIPI DI DATO

COSIMO LANEVE

`cosimo.laneve@unibo.it`

CORSO 00819 – PROGRAMMAZIONE

ARGOMENTI (SAVITCH: CAPITOLO 2, SEZIONI 2.1, 2.2, 2.3)

1. identificatori
2. dichiarazioni
3. tipi di dato
4. assegnamenti
5. espressioni
6. type safety
7. esercizi

IDENTIFICATORI

gli **identificatori** (o variabili) sono **nomi simbolici** creati dal programmatore ed associati ad un *valore*

regola: gli identificatori possono essere **sequenze** di **lettere**, **cifre**, e il simbolo “_”, che iniziano con una lettera oppure con “_”

esempi: `x_DC18` `X27` `un_identificatore`

quale è la grammatica? _ è un identificatore?

attenzione:

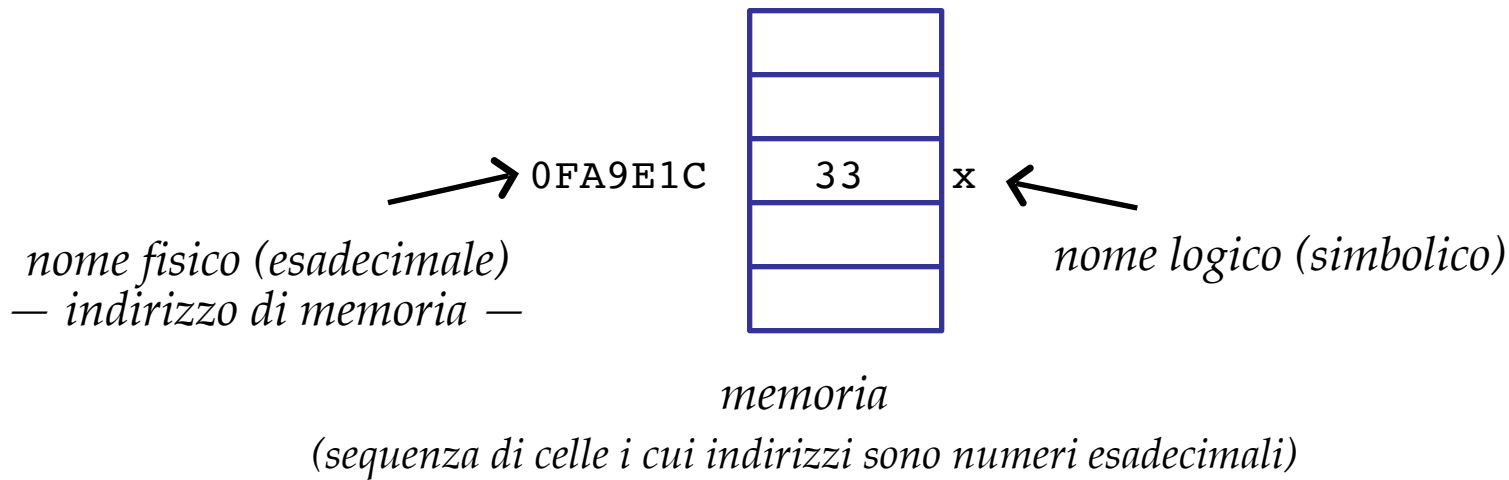
1. C++ è sensibile al tipo dei caratteri (se minuscolo o maiuscolo)

`un_ide` `UN_IDE` `Un_IdE` sono differenti

2. non si possono usare le parole chiavi (`int`, `float`, `double`,...) nè gli identificatori standard (`main`, `while`,...)

IDENTIFICATORI/NOMI LOGICI E NOMI FISICI

in pratica: un identificatore è il **nome associato ad una cella di memoria** utilizzata per contenere valori:



osservazione: il programmatore non può conoscere il nome fisico della cella di memoria perchè tale nome è noto soltanto al tempo di esecuzione

come si definiscono i nomi simbolici?

LE DICHIARAZIONI DEGLI IDENTIFICATORI

le dichiarazioni sono una parte di programma che comunica al compilatore

- * gli identificatori utilizzati
- * il tipo dei valori da memorizzare in ogni identificatore

esempio: `double kms, miles ;`

- * gli identificatori sono `kms` e `miles`
- * il tipo è `double` (**numeri reali**)

le liste sono non-vuote!

sintassi:

```
int lista_identificatori ;           // interi
double lista_identificatori ;       // reali
char lista_identificatori ;         // caratteri
```

altri esempi:

```
int prof, lunghe ;
char iniz_nome, iniz_cognome ;
double x, y, z ;     (ricordarsi delle " , " e del " ; ")
```

LE DICHIARAZIONI DEGLI IDENTIFICATORI/EFFETTI

le dichiarazioni servono ad **allocare la memoria sufficiente** a contenere i valori utilizzati dal programma

le dichiarazioni non hanno alcun effetto "visibile"

perchè si dichiara il tipo di informazione di un identificatore?

TIPI DI DATO

i valori manipolati dai programmi sono suddivisi in insiemi disgiunti, detti **tipi di dato**

i tipi di dato servono a ottimizzare l'uso della memoria

esempi di tipi di dato:

- * i numeri interi (`int`) occupano **4 byte** di memoria
- * i numeri reali (`double`) occupano **8 byte** di memoria
- * i caratteri (`char`) occupano **1 byte** di memoria

i tipi di dato servono anche a **ottimizzare** l'uso del processore: le operazioni sono suddivise a secondo della collezione di valori a cui si applicano

un tipo di dato è un insieme di valori e un insieme di operazioni definite su quei valori

TIPI DI DATO/INT

valori: sono il sottoinsieme degli interi che è possibile memorizzare in k byte


(di solito k=4: compresi tra $\pm 2.147.483.647$)

esempi: -16 0 3257 21

operazioni:

- * **memorizzare un intero** in una variabile di tipo `int`: `x= 27;`
- * effettuare **operazioni aritmetiche** (somma, differenza, moltiplicazione, divisione, resto ...) tra due interi: `5+4` `4/2` `5%2`
`4*7`
- * **confrontare due interi**: `5>4` `5==4` `5!=4` `5>= 4`
- * **valore assoluto** `abs (-3)`

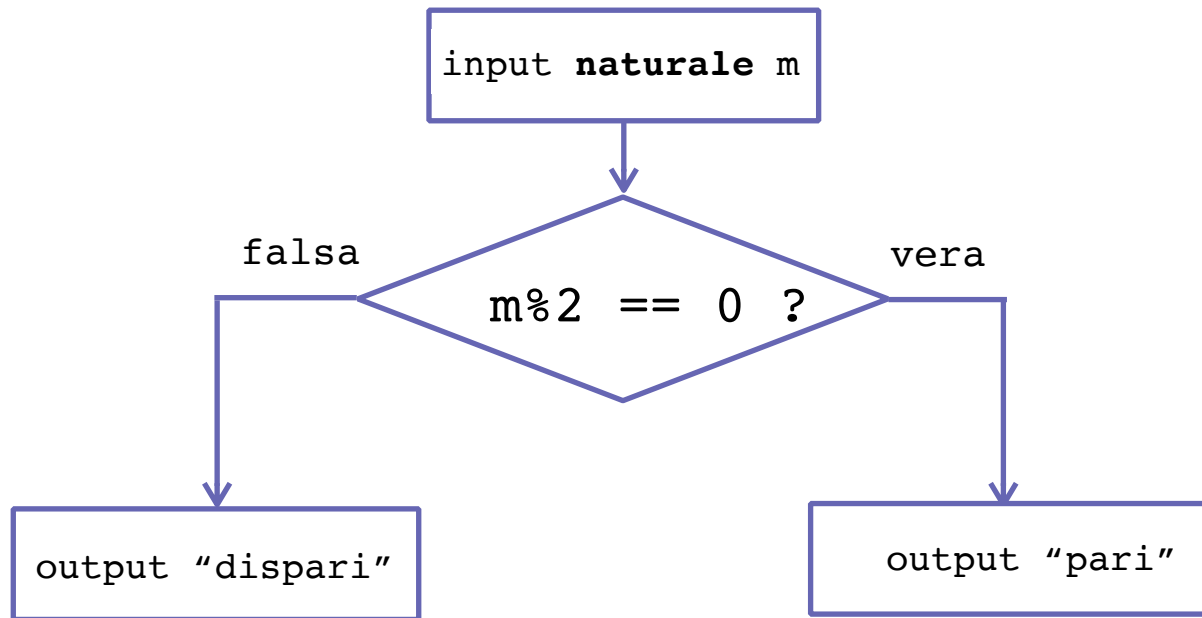
COMMENTI SULL'OPERAZIONE %

- * l'operatore di resto può essere applicato **solo a numeri interi**
 - altrimenti si ottiene un errore di tipo
- * l'operatore di resto **può essere derivato**: $n \% d$
coincide con $n - (n/d * d)$

perchè questa è la divisione intera
- * è comunque **miglior usare** l'operatore predefinito
- * se uno dei due operandi è negativo il risultato **dipende dall'implementazione**
- * se il secondo operando (divisore) è 0 si ha un **errore**

TEST DI PARITÀ

scrivere un programma che dato un numero intero scrive 0 se il numero è pari e 1 se il numero è dispari

algoritmo di qualche lezione fa

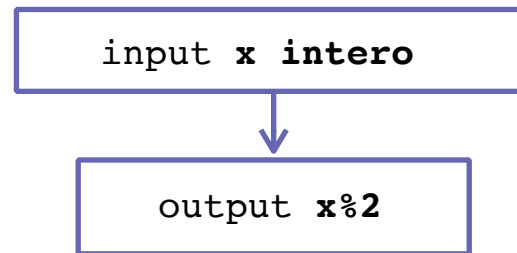


non va bene perchè adesso
l'output è 0 oppure 1

TEST DI PARITÀ

scrivere un programma che dato un numero intero scrive 0 se il numero è pari e 1 se il numero è dispari

algoritmo



implementazione

```
int main() {  
    int x ;  
    cin >> x ;  
    cout << x%2 ;  
    return(0) ;  
}
```

cin >> x prende in input un intero e lo memorizza in **x**

cout << e output di una espressione **e**

TIPI DI DATO/DOUBLE

valori: sono un'astrazione dei reali: alcuni reali sono troppo grandi o troppo piccoli, mentre altri non possono essere rappresentati in modo preciso

esempi:	notazione decimale	notazione scientifica
	3.1415	31.415e-1
	0.000016	0.16E-4
	120.0	12e1

valori che **non sono double**:

150	<i>// non c'è il punto</i>
3,45	<i>// la virgola non è consentita</i>
2e.3	<i>//.3 non è un esponente valido</i>
13e	<i>// manca l'esponente</i>

TIPI DI DATO/`DOUBLE`

operazioni:

- * **memorizzare** un reale in una variabile di tipo `double`
- * **operazioni aritmetiche** (somma, differenza, moltiplicazione, divisione, ...)
- * **confronto**
- * **operazioni di libreria** (es. `pow`, `log`, `sqrt`, ...):
includere `cmath`

esempi:

`5.1+4.0`

`pow(3.0, 2.0)`

`3.1>=0.3`

`log(4.1)`

TIPI DI DATO/DOUBLE/CAST

conversione interi → reali:

laddove può esserci un reale può comparire un intero

$4 / 2.3 \quad \rightarrow \quad 4.0 / 2.3$

$\log(4) \quad \rightarrow \quad \log(4.0)$

conversione reali → interi: il reale è troncato.

$4.6 \quad \rightarrow \quad 4$

il **cast** può essere anche esplicito:

$(\text{int}) 4.6 \quad \rightarrow \quad 4$

$(\text{double}) 4 \quad \rightarrow \quad 4.0$

TIPI DI DATO/CHAR

valori: sono i singoli caratteri (lettere, cifre, simboli speciali) ogni valore di tipo char è racchiuso da apostrofi

esempi: 'A' 'b' '7' ';'

operazioni:

* **memorizzare un carattere in una variabile** di tipo char

* **confrontare** due caratteri

esempi: 'A' > 'b' 'a' >= '@' 'a' == '@'

TIPI DI DATO/BOOL

valori: `true` e `false`

operazioni:

* `&&` (and logico), `||` (or logico), `!` (not logico)

* risultati dei confronti

esempi: `true||false` `('A'>'b') && (! (3>4))`

osservazione: il tipo `bool` non esiste in C

TIPI DI DATO/BOOL – TABELLE DI VERITÀ

tabella di verità di &&

op1	op2	op1 && op2
false	false	false
false	true	false
true	false	false
true	true	true

tabella di verità di ||

op1	op2	op1 op2
false	false	false
false	true	true
true	false	true
true	true	true

tabella di verità di !

op1	!op1
false	true
true	false

ISTRUZIONE DI ASSEGNAMENTO

memorizza un valore o il risultato di un calcolo in una variabile

sintassi: *variabile = espressione ;*

esempio:

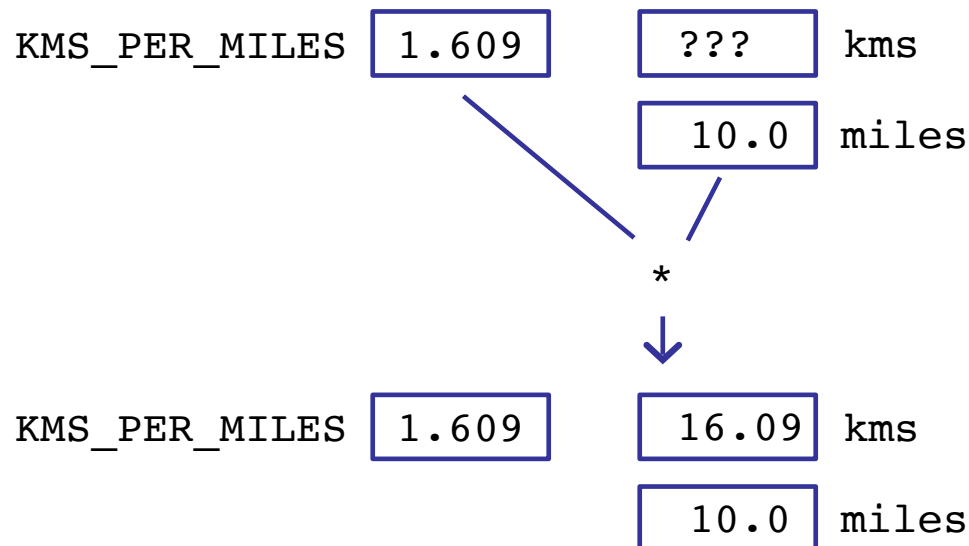
```
kms = KMS_PER_MILE * miles ;
```

ISTRUZIONE DI ASSEGNAMENTO

esempio: `kms = KMS_PER_MILE * miles ;`

- * **calcola il valore** dell'espressione `KMS_PER_MILE*miles`
- * se il calcolo della espressione **non produce errori** (non è questo il caso perchè "*" è totale)
- * il valore è **assegnato** alla variabile `kms`

* **esecuzione:**

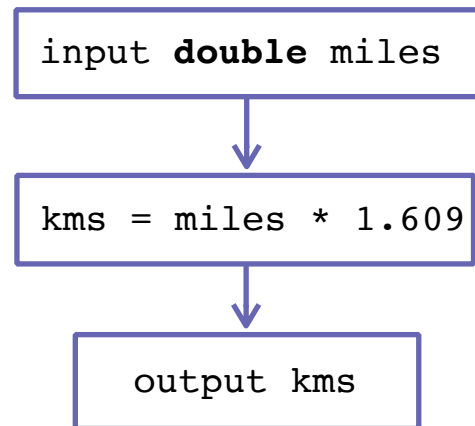


SCRIVERE UN PROGRAMMA CHE CONVERTE MIGLIA IN KM

il fattore di conversione è:

$$1 \text{ miglio} = 1.609 \text{ km}$$

algoritmo:



PROGRAMMA DI CONVERSIONE MIGLIA IN KM

```
#include <iostream>
using namespace std;
```

```
#define KMS_PER_MILE 1.609
```

dichiarazione di costante

```
int main() {
```

```
    double kms, miles ;
```

dichiarazioni di variabili
di tipo double

```
    cout << "distanza in miglia?> ";
```

```
    cin >> miles;
```

```
    kms = KMS_PER_MILE * miles;
```

assegnamento

```
    cout<<"la corrispondente distanza in km e` : ";
```

```
    cout << kms;
```

```
    return(0);
```

```
}
```

ESPRESSIONI

una espressione è una sequenza di operazioni che restituiscono un valore

una *espressione* può essere:

- * una **variabile**
- * una **costante**
- * (*una* **chiamata di funzione**)
- * una **combinazione** di variabili e costanti (e chiamate di funzioni) connesse da operatori (ad esempio $+$, $-$, $*$, $/$, $\%$)

esempi: $(5-2) - 4$ $y + (x * 5)$ $7 / (2 * x)$

VALUTAZIONE DELLE ESPRESSIONI

problema: quale è il valore di queste espressioni?

* $5 - 2 - 4$

* $4 + 3 - 2$

... vedi regole di precedenza degli operatori

consiglio: in caso di incertezza, **utilizzare le parentesi** per specificare l'ordine di valutazione

ORDINE DI VALUTAZIONE DELLE ESPRESSIONI

l'ordine di valutazione delle espressioni è fissato da:

- * **parentesi**
- * **precedenza tra operatori** (vedere tabella di precedenza di seguito)
- * operatori con stessa precedenza sono valutati da sinistra verso destra se binari, da destra verso sinistra se unari
- * l'ordine di valutazione degli operandi di un operatore binario **dipende dall'implementazione** (pensate a chiamate di funzioni definite dall'utente)

TABELLA DI PRECEDENZA DEGLI OPERATORI

!	+	-	&	*	(op. unari)	<i>precedenza più alta</i>
					per puntatori	
*	/	%			prodotto, divisione e resto	
+	-					
<	<=	>=	>			
==	!=					
&&						
						<i>precedenza più bassa</i>

VALUTAZIONE DELLE ESPRESSIONI: ESEMPI

* secondo quanto detto

$3 * 4 + 5$ viene valutato come $(3 * 4) + 5$

$10 - 3 - 2$ viene valutato come $(10 - 3) - 2$

* se voglio l'altro ordine di valutazione devo scrivere

$3 * (4 + 5)$ e $10 - (3 - 2)$

* **esercizio:** valutare le espressioni seguenti

$- 3 + 5$

$- - 3 * 4 == 12$

$3 > 2 \ \&\& \ 4 != 4$

ESPRESSIONI/TIPO

quale è il tipo di una espressione?

risposta: è determinato dalle **operazioni** e dal **tipo degli operandi**

esempio: `x + y`

- * se entrambi `x` ed `y` sono di tipo `int` allora l'espressione ha tipo `int`
- * se `x` o `y` hanno tipo `double` l'espressione ha tipo `double` (gli operatori aritmetici `+`, `-`, `*`, `/` sono overloaded)

attenzione: in `C++`

- * è possibile scrivere `3+'a'` oppure `3+true` oppure `3&&true`
- * noi non le scriveremo MAI

TYPE SAFETY

nei linguaggi di programmazione i tipi sono utilizzati per rilevare errori del programmatore

***type safety** = ogni entità deve essere usata in accordo con il suo tipo*

- * una variabile può essere usata solo **dopo** che è stata dichiarata
- * solamente le operazioni definite per il tipo dichiarato per la variabile **possono essere applicate** ad essa
- * ogni operazione (**totale**) applicata correttamente **ritorna un valore valido**

il compilatore riporta ogni violazione (in un ambiente di sviluppo ideale)

TYPE SAFETY

violazioni: 4.3 % 2

la **type safety** è un proprietà molto importante

cercate sempre di scrivere programmi type-safe

il compilatore è il vostro miglior amico quando programmate

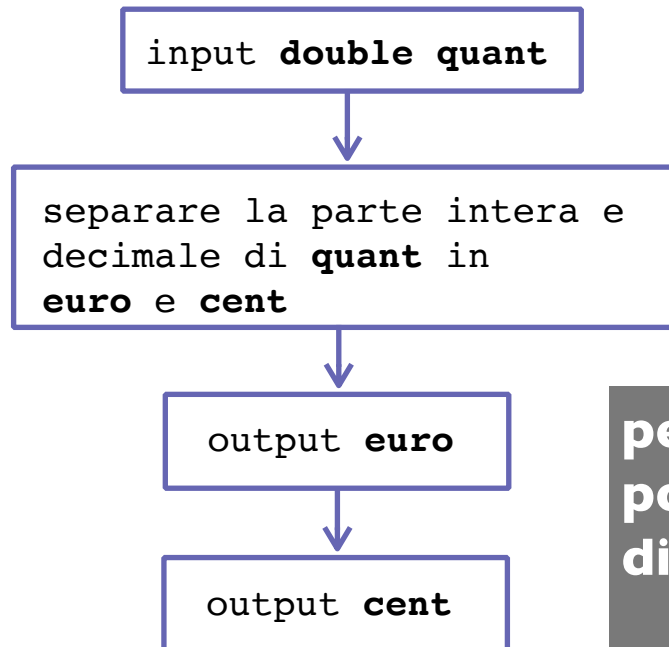
- * dubitate sempre della correttezza del programma quando non compila
- * in questo corso si darà particolare attenzione alla type safety: tutti i nostri programmi saranno type-safe

TIPI DI DATO/DOUBLE: PROBLEMI

attenzione ai problemi coi floating point!

scrivere un programma che prende una quantità in euro e stampa gli euro e i centesimi separatamente

algoritmo:



per separare la parte decimale
posso usare una variabile **euro**
di tipo **int** con l'assegnamento
`euro = quant ;`

TIPI DI DATO/DOUBLE: PROBLEMI

problemi con l'implementazione:

```
int main() {  
    double quant ;  
    int euro, cent ;  
    cout<<"scrivi la quantita " ;  
    cin >> quant ;  
    euro = quant ;  
    cout << "euro: " << euro << endl ;  
    cent = 100*(quant - euro) ;  
    cout << "centesimi: " << cent ;  
    return (0) ;  
}
```

* funziona con input 1.3

* **non** funziona con input 1.2

**le cose cambiano se cent
è dichiarato double**

TIPI DI DATO/DOUBLE: PROBLEMI

vediamo cosa accade:

**in quant viene memorizzato
1.199999999999**

```
int main() {  
    double quant ;  
    int euro, cent ;  
    cout<<"scrivi la quantita " ;  
    cin >> quant ;  
    euro = quant ;  
    cout << "euro: " << euro << endl;  
    cent = 100*(quant - euro) ;  
    cout << "centesimi: " << cent ;  
    return(0) ;  
}
```

// input 1.2
// euro = 1
// stampa 1
// cent = 19.9999
// stampa 19

ESERCIZI

1. dato questo frammento di codice

```
char x, y ;  
cin >> x >> y ;  
...  
cout << x << y ;
```

scrivere una sequenza di comandi che scambia il valore di due identificatori (quando gli identificatori sono `int` o `double`, si può fare senza un terzo identificatore...)

2. scrivere un programma che prende in input tre reali e li stampa in modo invertito

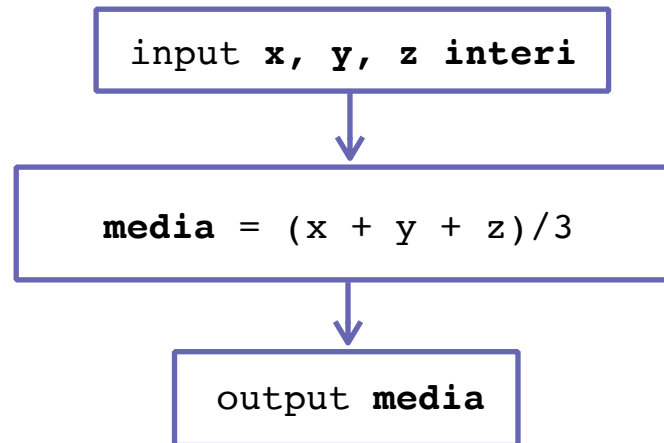
3. scrivere un programma che calcola l'area di un cerchio dato il raggio

– dichiarare le variabili usate

CASO DI STUDIO: MEDIA TRA INTERI

problema: scrivere un programma che prende in input tre interi e stampa il loro valor medio

algoritmo:



implementazione: problemi

1. oltre alle tre variabili intere, si utilizza una variabile **media**
2. quale è il tipo di **media**?
3. cosa succede se scrivo `media = (x + y + z)/3 ;`

MEDIA: OSSERVAZIONI

- * l'operatore di divisione si comporta in modo diverso tra interi e tra double
 - $(x+y+z)/3$ è diverso da $(x+y+z)/(\text{double})3$
- * il mancato uso del cast produce quindi risultati non precisi
- * il cast in quanto operatore unario ha precedenza sulla divisione
 - si può scrivere $(x+y+z)/(\text{double})3$ invece di $(x+y+z)/((\text{double})3)$
- * si può evitare di usare il cast con $(x+y+z)/3.0$

IMPLEMENTAZIONE DELLA MEDIA

```
int main() {  
    int x, y, z;  
    double media ;  
    cout << "il primo intero? > " ;  
    cin >> x ;  
    cout << "il secondo intero? > ";  
    cin >> y ;  
    cout << "il terzo intero? > ";  
    cin >> z ;  
    media = (x+y+z) / (double)3 ;  
    cout << "la media e` > " << media ;  
    return(0) ;  
}
```

COSTANTI

- * non è opportuno usare per pi-greco il valore numerico
 - si presta a errori
 - rende il programma poco leggibile (cos'è $6,672e-11$?)
- * non è opportuno usare per pi-greco una variabile
 - non vorremo mai modificare il valore di pi-greco
- * le costanti si dichiarano come le variabili, aggiungendo il prefisso **const**

```
const double pi = 3.14, e = 2.71;
```

- * oppure si dichiarano con

```
#define KMS_PER_MILE 1.609
```

INIZIALIZZAZIONE

- * non è possibile cambiare il valore di una costante (il compilatore dà errore)
- * per questo motivo le costanti devono essere inizializzate al momento della dichiarazione
- * possiamo usare l'inizializzazione anche per le variabili

esempio: `int age = 33;`

- * verificare sempre che una variabile sia inizializzata prima di essere usata. L'inizializzazione può avvenire
 - nella dichiarazione
 - tramite assegnamento
 - tramite input

C++ VS MATEMATICA

- * l'operatore `=` non è l'uguaglianza matematica, ma un **assegnamento**
- * il segno di prodotto non può essere sottointeso: $y = 2x$ è errato
- * per delimitare gli argomenti di una frazione si usano le parentesi: $(3 * 5) / 2$
- * in C++ posso avere due operatori consecutivi, se il secondo è unario: $4 * -2$

FUNZIONI DI LIBRERIA

* il C++ è dotato di librerie per calcolare le funzioni matematiche di uso più comune

* includendo `<cmath>` abbiamo:

- `double abs(double)`
- `double sqrt(double)`
- `double pow(double, double)`
- `double cos(double)`
- `double sin(double)`
- . . .

* includendo `<cstdlib>` abbiamo:

- `int rand();`
- `int srand(int);`
- `RAND_MAX`

PROGRAMMA PER NUMERI PSEUDO-CASUALI

scrivere un programma che genera un numero random tra 0 e 89

```
#include <iostream>
#include <cstdlib>
#include <ctime>
```

```
using namespace std;
```

inizializza la funzione
di generazione di numeri
pseudo-casuali

```
int main() {
    srand( time(0) ) ;
    cout << rand() % 90 ;
    return(0) ;
}
```

restituisce il tempo

funzione di generazione
di numeri pseudo-casuali

ESERCIZI

1. scrivere un programma che prende in input un prezzo in euro (inclusi centesimi) e stampa quante e quali monete sono necessarie per pagarlo (con un numero minimo di monete)

```
esempio:      input: 15.74
               output: monete da 2 euro: 7
                      monete da 1 euro: 1
                      monete da 50 cent: 1
                      monete da 20 cent: 1
                      monete da 2 cent: 2
```

2. scrivere un programma che prende in input la lunghezza di due cateti di un triangolo rettangolo e stampa la lunghezza dell'ipotenusa

INPUT/OUTPUT

- * i costrutti di input/output sono fondamentali per interagire col vostro programma
- * in C++ un modo semplice e potente per fare input output è dato dagli stream **cin** e **cout**
 - uno **stream** è una sequenza di caratteri
 - per usare gli stream ricordarsi di includere `<iostream>`
- * si interagisce con gli stream tramite gli operatori `<<` e `>>`
 - capiremo il funzionamento di questi operatori più avanti (operatori overloaded), per il momento ci accontenteremo di capire come funzionano
- * in C++ esistono anche altri modi per fare input/output, ma non li vedremo in questo corso

OPERATORE DI OUTPUT <<

- * l'operatore << invia valori al canale di output specificato (noi useremo solo **cout**)

sintassi: `cout << exp ; // calcola exp e scrive il
 // valore di exp in cout`

- * l'operatore << è in grado di stampare dati di (quasi) tutti i tipi base

```
int x=5 ;  
cout << "x=" ;  
cout << x ;
```

- * possiamo **comporre** l'operatore << per stampare più valori con un'unica istruzione

```
cout << "x=" << x ;
```

OPERATORE DI OUTPUT <<

- * l'operatore << associa a sinistra (e restituisce lo stream stesso)

```
(cout << "x=") << x ;
```

- * **il livello di precedenza è inferiore a quello degli operatori aritmetici (ma non di quelli logici),** quindi permette l'uso di espressioni aritmetiche senza parentesi:

```
cout << "x+y=" << x+y ;
```

SEQUENZE DI ESCAPE

- * sono usate per inserire caratteri speciali
- * sono composte da backslash (\) seguito da un codice speciale
 - `\n` nuova linea
 - `\t` tab
 - `\\` backslash
- * nel programma `Hello World` su `Eclipse` trovate il manipolatore `endl` dopo l'operazione di output:
 - `endl` inserisce `\n` nello stream, e stampa a video eventuali caratteri rimasti nello stream

OPERATORE DI INPUT >>

- * l'operatore >> riceve valori da un canale di input (noi useremo solo `cin`) salvandoli nel secondo argomento

sintassi: `cin >> lhs-exp ; // legge da cin il
// valore in lhs-exp`

■ per il momento `lhs-exp` \equiv `identificatore`

- * possiamo comporre l'operatore >> per leggere più valori con un'unica istruzione

```
int x,y ;  
cin >> x >> y ;
```

- * **associatività e precedenza di >> sono come per <<, e anche lui restituisce lo stream**

OPERATORE DI INPUT >>

l'operatore >> ha un comportamento diverso a seconda del tipo di dato

esempio:

```
char x ;  
int y ;  
cin >> x >> y ;
```

- * poichè **x** è una variabile **char**, **cin** legge un carattere
- * poichè **y** è una variabile **int**, **cin** legge finchè trova caratteri numerici validi
- * se un identificatore è una variabile a virgola mobile legge finchè trova caratteri validi per valori a virgola mobile
- * eventuali **spazi o a capo iniziali** non vengono considerati

INPUT/OUTPUT: ESERCIZI

1. scrivere un programma che chieda in input la vostra età e il vostro sesso (come carattere M/F) e li stampi a video
2. riscrivere il programma precedente usando una sola volta **cin** e una sola volta **cout**
3. scrivere un programma che presi in input la base e l'altezza di un triangolo ne stampi l'area
4. scrivere un programma che prende in input 3 numeri interi e produce in output una tabella con la differenza dei numeri a due a due.

esempio: se i numeri sono 2, 7, 3 deve stampare

	2	7	3
2	0	-5	-1
7	5	0	4
3	1	-4	0

SPAZIATURA: ESEMPI

- * il seguente programma è corretto secondo il compilatore, ma non leggibile

```
#include <iostream>
using namespace std;
int main(){cout<<"!!!Hello World!!!"<<endl;return 0;}
```

- * altri esempi di spaziature

ok

main

KMS_PER_MILE

no

ma in

KMS PER MILE

COMMENTI

- * sono informazioni inserite per aumentare la leggibilità del programma
 - non hanno alcun effetto sul risultato della compilazione
 - ma hanno effetto sulla comprensione del programma
- * i commenti vanno inseriti ad esempio
 - in corrispondenza dei prototipi di funzioni, per descriverne **precondizioni** e **postcondizioni**
 - vicino alle dichiarazioni di variabile/costante, per spiegarne l'utilità
 - prima di un passo **complesso** dell'algoritmo, per descriverlo
- * troppi commenti o commenti inutili sono dannosi

COMMENTI

* sintassi

```
/* testo del commento */  
// testo del commento
```

* un commento può essere inserito dovunque possa essere inserito uno spazio:

```
double raggio; /* raggio del cerchio*/  
/* commento su più righe che descrive  
* l'obiettivo di una parte particolarmente  
* complessa dell'algoritmo  
*/
```

* esempio di **commento inutile**

```
i = i + 1; /* somma 1 al valore di i */
```

SPAZIATURE E INDENTAZIONI

- * il compilatore ignora gli spazi tra parole e simboli, e tratta gli "a capo" come spazi
- * gli spazi devono essere utilizzati per rendere i programmi chiari e leggibili
 - inserire uno spazio dopo la virgola
 - inserire uno spazio prima e dopo gli operatori (+, *, ...)
 - scrivere un'istruzione per riga (se un'istruzione è troppo lunga, può essere suddivisa su più righe)
 - indentare il corpo delle funzioni
 - separare le sezioni di un programma mediante righe vuote
- * non è possibile inserire spazi:
 - all'interno di un delimitatore di commento
 - all'interno di un identificatore o di una parola riservata