Struttura dei programmi C

- Nel semplice programma che abbiamo appena analizzato possiamo già vedere la struttura generale di un programma C.

```
/* DIRETTIVE DI COMPILAZIONE */
#include <stdio.h>
 main() {

 /* PARTE DICHIARATIVA */
  int base;
  int altezza;
  int area;

 /* PARTE ESECUTIVA */
  base = 3;
  altezza = 4;
  area = base * altezza;
  printf("Area: %d\n", area);
}
```

Un programma C deve contenere nell'ordine:

- Una parte contenente direttive per il compilatore. Nel nostro programma la direttiva

```
#include <stdio.h>
```

- l'identificatore predefinito `main` seguito dalle parentesi `()`.
- due parti racchiuse tra parentesi graffe
  - la parte dichiarativa. Nell'esempio:

```
int base;
int altezza;
int area;
```

  - la parte esecutiva. Nell'esempio:

```
base = 3;
altezza = 4;
area = base * altezza;
printf("Area: %d\n", area);
```

La parte dichiarativa

- È posta prima della codifica dell'algoritmo e obbliga il programmatore a dichiarare i nomi simbolici che saranno presenti nello stato e di cui farà uso nella parte esecutiva. Contiene i seguenti elementi:
  - la sezione delle dichiarazioni di variabili;
  - la sezione delle dichiarazioni di costanti.
- Le dichiarazioni:
  - rendono più pesante la fase di costruzione dei programmi, ma
  - consentono di individuare e segnalare errori in fase di compilazione.

Esempio:

```
int x;
int alfa;
alfa = 0;
x=alfa;
alba=alfa+1;
```

- Nell'ultima linea abbiamo erroneamente scambiato una **b** con una **f**
  $\Longrightarrow$ il compilatore individua `alba` come variabile non dichiarata.

Dichiarazioni di variabili

- Abbiamo già visto esempi di dichiarazioni di variabili.

```
float x;
int base;
int altezza;
```

- Ad ogni variabile viene attribuito, al momento della dichiarazione, un tipo

  $\Longrightarrow$ specifica l'insieme dei valori che la variabile può assumere

- La dichiarazione può anche attribuire un valore iniziale alla variabile (inizializzazione)

```
int x = 0;
```

- Variabili dello stesso tipo possono essere dichiarate contemporaneamente

```
int base, altezza, area;
```

(ma inizializzate singolarmente)

Esempio: `int x, y, z=0;` solo `z` è inizializzata a `0`.

Area di un rettangolo di dimensioni lette da tastiera

```
#include <stdio.h>

main()
{
  int base, altezza, area;

  printf("Immetti base del rettangolo e premi INVIO\n");
  scanf("%d", &base);
  printf("Immetti altezza del rettangolo e premi INVIO\n");
  scanf("%d", &altezza);

  area = base * altezza;

  printf("Area: %d\n", area);
}
```

Nuova istruzione: `scanf("%d", &base);`

- `scanf` è la funzione duale di `printf`
- legge da input (tastiera) un valore intero e lo assegna alla variabile `base`
- `"%d"` è la stringa di controllo del formato (in questo caso viene letto un intero in formato decimale)
- "`&`" è l'operatore di indirizzo
  - `&base` indica (l'indirizzo del)la locazione di memoria associata a `base`
  - `scanf` memorizza in tale locazione il valore letto
- quando viene eseguita `scanf` il programma si mette in attesa che l'utente immetta un valore. Quando l'utente digita Invio
  1. la sequenza di caratteri immessa viene convertita in un intero (formato `%d`) e
  2. l'intero ottenuto viene assegnato alla variabile `base` (viene cioè scritto nella/e cella/e di memoria a partire dall'indirizzo passato a `scanf`)

N.B. il precedente valore della variabile `base` va perduto

Esempio di esecuzione

- Vediamo cosa avviene durante l'esecuzione (indichiamo in rosso ciò che l'utente digita e in particolare con $\hookleftarrow$ il tasto Invio).

```
Immetti base del rettangolo e premi INVIO
5 ↩
Immetti altezza del rettangolo e premi INVIO
4 ↩
Area: 20
```

Dichiarazioni di costanti (variabili *read-only*)

- Una dichiarazione di costante crea un'associazione non modificabile $\Longrightarrow$ associa in modo permanente un valore ad un identificatore.

  Esempio:

  ```
  const float PiGreco=3.14;
  const int N=100;
  ```

- L'associazione tra il nome `PiGreco` ed il valore `3.14` non può essere modificata durante l'esecuzione.
- Come per le dichiarazioni di variabili, più costanti dello stesso tipo possono essere dichiarate insieme

  Esempio:

  ```
  const float PiGreco=3.14, e=2.718;
  const int N=100, M=200;
  ```

- **N.B.** cosa succede quando si modifica una variabile read-only non è specificato dallo standard ANSI C, dipende dal compilatore.

Uso di costanti

- Con la dichiarazione `const float PiGreco=3.14;`
  l'istruzione
  `AreaCerchio=PiGreco*RaggioCerchio*RaggioCerchio;`
  è equivalente a
  `AreaCerchio=3.14*RaggioCerchio*RaggioCerchio`
- Maggiore leggibilità dei programmi, dovuta all'uso di nomi simbolici
- Maggiore adattabilità dei programmi che usano costanti

  Esempio:
  Per aumentare la precisione, basta cambiare la dichiarazione in
  `const float PiGreco = 3.1415;`

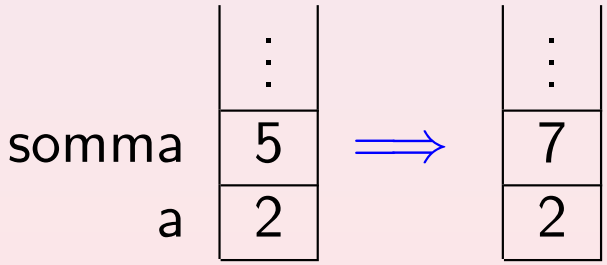
  Senza l'uso della costante si dovrebbero rimpiazzare nel codice tutte le occorrenze di `3.14` in `3.1415` ...

Assegnamento

- Ricordiamo che l'esecuzione di `x = exp` corrisponde a:
  1. valutare il valore dell'espressione `exp` a destra di "=" (usando i valori correnti delle variabili);
  2. assegnare poi tale valore alla variabile `x` a sinistra di "=".

Esempio:

```
somma = 5;
a = 2;
somma = somma + a;
```



Esempio:

	a	b
`int a, b;`	?	?
`a = 2;`	2	?
`b = 3;`	2	3
`a = b;`	3	3
`a = a + b;`	6	3
`b = a + b;`	6	9

Osservazioni sull'assegnamento

- **Attenzione:** A sinistra di "=" ci deve essere un identificatore di variabile

  $\Longrightarrow$ denota la corrispondente associazione modificabile nello stato.

  Esempio: Quali istruzioni sono corrette e quali no?

  `a = a;` SI corretta (anche se poco significativa . . .)

  `a = 2 * a;` SI corretta (il valore associato ad `a` viene raddoppiato)

  `5 = a;` NO, `5` non denota una associazione modificabile nello stato ma un valore costante

  `a + b = c;` NO, `a+b` è un'espressione, non una variabile!

- Tutti i programmi che abbiamo visto fino ad ora prevedono istruzioni che vengono eseguite in sequenza. Ma come si fa ad esempio a leggere 2 interi e a stampare il valore massimo?
- Cioè come si fa ad eseguire un istruzione solo se si verifica una determinata condizione?
- la risposta sono le istruzioni condizionali (esistono praticamente identiche in quasi tutti i linguaggi di programmazione).
- Per poter definire le istruzioni condizionali è necessario poter esprimere le condizioni.
- Le condizioni si esprimono attraverso le espressioni booleane.

Espressioni booleane

- In C non esiste un tipo Booleano $\Longrightarrow$ si usa il tipo **int** :

  falso $\Longleftrightarrow$ 0

  vero $\Longleftrightarrow$ 1 (in realtà qualsiasi valore diverso da 0)

  Esempio: $2 > 3$ ha valore 0 (ossia falso)

  $5 > 3$ ha valore 1 (ossia vero)

- **Operatori relazionali del C**
  - `<`, `>`, `<=`, `>=` (minore, maggiore, minore o uguale, maggiore o uguale) — priorità alta
  - `==`, `!=` (uguale, diverso) — priorità bassa

  Esempio: `temperatura <= 0` `velocita > velocita_max`
  `voto == 30` `anno != 2000`

Operatori logici

- In ordine di priorità:
  - `!` (negazione) — priorità alta
  - `&&` (congiunzione)
  - `||` (disgiunzione) — priorità bassa

Semantica:

a	b	!a	a && b	a || b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

0 . . . falso

1 . . . vero (qualsiasi valore $\neq 0$)

Esempio:

`(a >= 10) && (a <= 20)` vero (`1`) se $10 \leq a \leq 20$

`(b <= -5) || (b >= 5)` vero se $|b| \geq 5$

- Le espressioni booleane vengono valutate da sinistra a destra:
  - con `&&`, appena uno degli operandi è falso, restituisce falso senza valutare il secondo operando
  - con `||`, appena uno degli operandi è vero, restituisce vero senza valutare il secondo operando
- **Priorità** tra operatori di diverso tipo:
  - not logico — priorità alta
  - aritmetici
  - relazionali
  - booleani (and e or logico) — priorità bassa

Esempio:

```
a+2 == 3*b || !trovato && c < a/3
```

è equivalente a

```
((a+2) == (3*b)) || ((!trovato) && (c < (a/3)))
```

I caratteri: il tipo char

Esempio:

```
char x, y, z;
x = 'a';
y = 'd';
```

- Posso valutare $(x \leq y)$.
- Cosa mi dice? Se x precede y nell'ordine alfabetico.

Esempio:

```
char ch1 = 'a';
char ch2 = 'c';
char ch3 = (char) ((ch1 + ch2)/2);
printf("%c'', ch3);
```

- Cosa stampa? Stampa il carattere `'b'`.

Come va usato il codice dei caratteri

- Convertiamo una lettera minuscola in maiuscolo:

  Esempio:

```
char lower = 'k';
char upper = (char) (lower - 'a' + 'A');
printf("%c", upper);
```

- Convertiamo un carattere numerico (una cifra) nell'intero corrispondente:

  Esempio:

```
char ch1 = '9';
int num = ch1-'0';
```

- Questi frammenti di programma sono completamente portabili (non dipendono dal codice usato per la rappresentazione dei caratteri).

Istruzione **if-else**

Sintassi:

```
if    (espressione)
      istruzione1
else  istruzione2
```

- `espressione` è un'espressione booleana
- `istruzione1` rappresenta il ramo then (deve essere un'unica istruzione)
- `istruzione2` rappresenta il ramo else (deve essere un'unica istruzione)

Semantica:

1. viene prima valutata `espressione`
2. se `espressione` è vera viene eseguita `istruzione1`
   altrimenti (ovvero se `espressione` è falsa) viene eseguita `istruzione2`

```
int temperatura;

printf("Quanti gradi ci sono?  "); scanf("%d", &temperatura);
if (temperatura >= 25)
  printf("Fa caldo\n");
else
  printf("Si sta bene\n");

printf("Arrivederci\n");
```

=> Quanti gradi ci sono? 30 ↩
Fa caldo
Arrivederci
=>

=> Quanti gradi ci sono? 18 ↩
Si sta bene
Arrivederci
=>

Istruzione **if**

- È un'istruzione **if-else** in cui manca la parte else.

Sintassi:

```
if (espressione)
   istruzione
```

Semantica:

1. viene prima valutata `espressione`
2. se `espressione` è vera viene eseguita `istruzione` altrimenti non si fa alcunché

Esempio:

```
int temperatura;
scanf("%d", &temperatura);
if (temperatura >= 25)
  printf("Fa caldo\n");
printf("Arrivederci\n");
```

=> 18 ↩
Arrivederci

=> 30 ↩
Fa caldo
Arrivederci

Blocco

- La sintassi di **if-else** consente di avere un'unica istruzione nel ramo then (o nel ramo else).
- Se in un ramo vogliamo eseguire più istruzioni dobbiamo usare un blocco.

  Sintassi:

```
{
    istruzione-1
    ...
    istruzione-n
}
```

- Come già sappiamo e come rivedremo più avanti, un blocco può contenere anche dichiarazioni.

Esempio: Dati mese ed anno, calcolare mese ed anno del mese successivo.

```
int mese, anno, mesesucc, annosucc;

if (mese == 12)
{
  mesesucc = 1;
  annosucc = anno + 1;
}
else
{
  mesesucc = mese + 1;
  annosucc = anno;
}
```

If annidati (in cascata)

- Si hanno quando l'istruzione del ramo then o else è un'istruzione **if o if-else.**

  Esempio: Data una temperatura, stampare un messaggio secondo la seguente tabella:

temperatura t	messaggio
$30 < t$	Molto caldo
$20 < t \leq 30$	Caldo
$10 < t \leq 20$	Gradevole
$0 < t \leq 10$	Freddo
$t \leq 0$	Molto freddo

```
if (temperatura > 30)
  printf("Molto caldo\n");
else if (temperatura > 20)
  printf("Caldo\n");
else if (temperatura > 10)
  printf("Gradevole\n");
else if (temperatura > 0)
  printf("Freddo\n");
else
  printf("Molto freddo\n");
```

Osservazioni:

- si tratta di un'unica istruzione **if-else**

```
if (temperatura > 30)
  printf("Molto caldo\n");
else ...
```

- non serve che la seconda condizione sia composta

```
if (temperatura > 30) printf("Molto caldo\n");
else /* il valore di temperatura e' <= 30 */
    if (temperatura > 20)
```

  Non c'è bisogno di una congiunzione del tipo
  `(t <= 30) && (t > 20)`
  (analogamente per gli altri casi).

- **Attenzione**: il seguente codice

```
if (temperatura > 30) printf("Molto caldo\n");
if (temperatura > 20) printf("Caldo\n");
```

  ha ben altro significato (quale?)

Ambiguità dell'else

```
if (a >= 0) if (b >= 0) printf("b positivo");
else printf("???");
```

- `printf("???")` può essere la parte **else**
  - del primo **if** $\Longrightarrow$ `printf("a negativo");`
  - del secondo **if** $\Longrightarrow$ `printf("b negativo");`
- L'ambiguità sintattica si risolve considerando che un **else** fa sempre riferimento all'**if** più vicino, dunque

```
if (a > 0)
  if (b > 0)
    printf("b positivo");
  else

    printf("b negativo");
```

- Perché un **else** si riferisca ad un **if** precedente, bisogna inserire quest'ultimo in un blocco

```
 if (a > 0)
  { if (b > 0) printf("b positivo"); }
else

  printf("a negativo");
```

Esercizio

Leggere un reale e stampare un messaggio secondo la seguente tabella:

gradi alcolici g	messaggio
$40 < g$	superalcolico
$20 < g \leq 40$	alcolico
$15 < g \leq 20$	vino liquoroso
$12 < g \leq 15$	vino forte
$10.5 < g \leq 12$	vino normale
$g \leq 10.5$	vino leggero

Esempio: Dati tre valori che rappresentano le lunghezze dei lati di un triangolo, stabilire se si tratti di un triangolo equilatero, isoscele o scaleno.

Algoritmo: `determina tipo di triangolo`

```
leggi i tre lati
confronta i lati a coppie, fin quando non
  hai raccolto una quantità di informazioni
  sufficiente a prendere la decisione
stampa il risultato
```

```
main()   {
float primo, secondo, terzo;

printf("Lunghezze lati triangolo ?  ");
scanf("%f%f%f", &primo, &secondo, &terzo);

if (primo == secondo) {
  if (secondo == terzo)
    printf("Equilatero\n");
  else
    printf("Isoscele\n");
}
else {
  if (secondo == terzo)
    printf("Isoscele\n");
  else if (primo == terzo)
    printf("Isoscele\n");
  else
    printf("Scaleno\n");
}     }
```

Esercizio

Risolvere il problema del triangolo utilizzando il seguente algoritmo:

```
Algoritmo: determina tipo di triangolo con conteggio
leggi i tre lati
confronta i lati a coppie contando
  quante coppie sono uguali
if le coppie uguali sono 0
  è scaleno
  else if le coppie uguali sono 1
    è isoscele
    else è equilatero
```

Iterazione determinata e indeterminata

- Le istruzioni iterative permettono di ripetere determinate azioni più volte:
  - un numero di volte fissato $\Longrightarrow$ iterazione determinata
    Esempio:
    ```
    fai un giro del parco di corsa per 10 volte
    ```
  - finchè una condizione rimane vera $\Longrightarrow$ iterazione indeterminata
    Esempio:
    ```
    finche' non sei sazio
    prendi una ciliegia dal piatto e mangiala
    ```

Istruzione **while**

Permette di realizzare l'iterazione in C.

Sintassi:

```
while (espressione)
      istruzione
```

- `espressione` è la guardia del ciclo
- `istruzione` è il corpo del ciclo (può essere un blocco)

Semantica:

1. viene valutata l'`espressione`
2. se è vera si esegue `istruzione` e si torna ad eseguire l'intero `while`
3. se è falsa si termina l'esecuzione del `while`

- Nota: se `espressione` è falsa all'inizio, il ciclo non fa nulla.

Iterazione determinata

Esempio: Leggere 10 interi, calcolarne la somma e stamparla.

- Si utilizza un contatore per contare il numero di interi letti.

```
int conta, dato, somma;
printf("Immetti 10 interi:  ");
somma = 0;
conta = 0;
while (conta < 10) {
  scanf("%d", &dato);
  somma = somma + dato;
  conta = conta + 1;
}
printf("La somma e' %d\n", somma);
```

Esempio: Leggere un intero N seguito da N interi e calcolare la somma di questi ultimi.

- Simile al precedente: il numero di ripetizioni necessarie non è noto al momento della scrittura del programma ma lo è al momento dell'esecuzione del ciclo.

```
int lung, conta, dato, somma;
printf("Immetti la lunghezza della sequenza ");
printf("seguita dagli elementi della stessa:  ");
scanf("%d", &lung);
somma = 0;
conta = 0;
while (conta < lung) {
  scanf("%d", &dato);
  somma = somma + dato;
  conta = conta + 1;
}
printf("La somma e' %d\n", somma);
```

Esempio: Leggere 10 interi positivi e stamparne il massimo.

- Si utilizza un massimo corrente con il quale si confronta ciascun numero letto.

```
int conta, dato, massimo;
printf("Immetti 10 interi:  ");
massimo = 0;
conta = 0;
while (conta < 10) {
  scanf("%d", &dato);
  if (dato > massimo)
    massimo = dato;
  conta = conta + 1;
}
printf("Il massimo e' %d\n", massimo);
```

Esercizio

Leggere 10 interi arbitrari e stamparne il massimo.

Istruzione **for**

- I cicli visti fino ad ora hanno queste caratteristiche comuni:
  - utilizzano una variabile di controllo
  - la guardia verifica se la variabile di controllo ha raggiunto un limite prefissato
  - ad ogni iterazione si esegue un'azione
  - al termine di ogni iterazione viene incrementato (decrementato) il valore della variabile di controllo

Esempio: Stampare i numeri pari da 0 a N.

```
i = 0; /* Inizializzazione della var.  di controllo */
while (i <= N) { /* guardia */
  printf("%d ", i); /* Azione da ripetere */
  i=i+2; /* Incremento var.  di controllo */
}
```

- L'istruzione **for** permette di gestire direttamente questi aspetti:

```
for (i = 0; i <= N; i=i+2)
  printf("%d", i);
```

Sintassi:

```
for (istr-1; espr-2; istr-3)
  istruzione
```

- `istr-1` serve a inizializzare la variabile di controllo
- `espr-2` è la verifica di fine ciclo
- `istr-3` serve a incrementare la variabile di controllo alla fine del corpo del ciclo
- `istruzione` è il corpo del ciclo

Semantica: l'istruzione **for** precedente è equivalente a

```
istr-1;
while (espr-2) {
  istruzione
  istr-3
}
```

Esempio:

```
for (i = 1; i <= 10; i=i+1)      ⟹   i: 1, 2, 3, ..., 10
for (i = 10; i >= 1; i=i-1)      ⟹   i: 10, 9, 8, ..., 2, 1
for (i = -4; i <= 4; i = i+2)    ⟹   i: -4, -2, 0, 2, 4
for (i = 0; i >= -10; i = i-3)   ⟹   i: 0, -3, -6, -9
```

- In realtà, la sintassi del **for** è

  ```
  for (espr-1; espr-2; espr-3)
    istruzione
  ```

  dove `espr-1`, `espr-2` e `espr-3` sono delle espressioni qualsiasi (in C anche l'assegnamento è un'espressione ...).
- È buona prassi:
  - usare ciascuna `espr-i` in base al significato descritto prima
  - non modificare la variabile di controllo nel corpo del ciclo
- Ciascuna delle tre `espr-i` può anche mancare:
  - i ";" vanno messi lo stesso
  - se manca `espr-2` viene assunto il valore vero
- Se manca una delle tre `espr-i` è meglio usare un'istruzione **while**

Esempio: Leggere 10 interi positivi e stamparne il massimo.

```
int conta, dato, massimo;
printf("Immetti 10 interi:  ");
massimo = 0;
for (conta=0; conta<10; conta=conta+1)
{
  scanf("%d", &dato);
  if (dato > massimo)
    massimo = dato;
}
printf("Il massimo e' %d\n", massimo);
```

Iterazione indeterminata

- In alcuni casi il numero di iterazioni da effettuare non è noto prima di iniziare il ciclo, perché dipende dal verificarsi di una condizione.

  Esempio: Leggere una sequenza di interi che termina con 0 e calcolarne la somma.

  Input: $n_1, \ldots, n_k, 0$ (con $n_i \neq 0$)

  Output: $\sum_{i=1}^{k} n_i$

```
int dato, somma = 0;
scanf("%d", &dato);
while (dato != 0) {
  somma = somma + dato;
  scanf("%d", &dato);
}
printf("%d", somma);
```

Istruzione **do-while**

- Nell'istruzione **while** la condizione di fine ciclo viene controllata all'inizio di ogni iterazione.
- L'istruzione **do-while** è simile all'istruzione **while**, ma la condizione viene controllata alla fine di ogni iterazione

Sintassi:

```
do
  istruzione
while (espressione);
```

Semantica: è equivalente a

```
istruzione
while (espressione)
  istruzione
```

$\Longrightarrow$ una iterazione viene eseguita comunque.

Esempio: Lunghezza di una sequenza di interi terminata da 0, usando **do-while**.

```
main() {
int lunghezza = 0; /* lunghezza della sequenza */
int dato; /* dato letto di volta in volta */
printf("Inserisci una sequenza di interi (0 fine seq.)\n");
do {
  scanf("%d", &dato);
  lunghezza=lunghezza+1;
} while (dato != 0);
printf("La sequenza e' lunga %d\n", lunghezza - 1);
}
```

- Nota: lo 0 finale non è conteggiato (non fa parte della sequenza, fa da terminatore)

Esempio: Leggere due interi positivi e calcolarne il massimo comun divisore.

$$\begin{aligned} MCD(12, 8) &= 4 \\ MCD(12, 6) &= 6 \\ MCD(12, 7) &= 1 \end{aligned}$$

- Sfruttando direttamente la definizione di MCD
  - osservazione: $1 \leq MCD(m,n) \leq min(m,n)$
    $\Longrightarrow$ si provano i numeri compresi tra 1 e $min(m,n)$
  - conviene iniziare da $min(m,n)$ e scendere verso 1

Algoritmo: stampa MCD di due interi positivi letti da tastiera

```
leggi m ed n
inizializza mcd al minimo tra m ed n
while mcd > 1 e non si e' trovato un divisore comune
{
  if mcd divide sia m che n
    si e' trovato un divisore comune
  else decrementa mcd di 1
}
stampa mcd
```

Osservazioni

- il ciclo termina sempre perché ad ogni iterazione
  - o si è trovato un divisore
  - o si decrementa mcd di 1 (al più si arriva a 1)
- per verificare se si è trovato il MCD si utilizza una variabile booleana (nella guardia del ciclo)
- Implementazione in C . . .

```
int m, n;                /* i due numeri letti */
int mcd;                /* il massimo comun divisore */
int trovato = 0;        /* var. booleana: inizialmente false */
if (m <= n)             /*inizializza mcd al minimo tra m e n*/
  mcd = m;
else
  mcd = n;
while (mcd > 1 && !trovato)
  if ((m % mcd == 0) && (n % mcd == 0))
    /* mcd divide entrambi */
    trovato = 1;
  else
    mcd = mcd -1;
printf("MCD di %d e %d:  %d", m, n, mcd);
```

Quante volte viene eseguito il ciclo?

- caso migliore: 1 volta (quando m divide n o viceversa)
  es. MCD(500, 1000)
- caso peggiore: *min*(m,n) volte (quando MCD(m,n)=1)
  es. MCD(500, 1001)
- l'algoritmo si comporta *male* se m e n sono grandi e MCD(m,n) è piccolo

Metodo di Euclide per il calcolo del MCD

- Permette di ridursi più velocemente a numeri più piccoli, sfruttando le seguenti proprietà:

  MCD(x, x) = x
  MCD(x, y) = MCD(x-y, y) se x>y
  MCD(x, y) = MCD(x, y-x) se y>x

- I divisori comuni di m ed n, con m>n, sono anche divisori di m-n.
  Es.: MCD(12, 8) = MCD(12-8, 8) = MCD(4, 8-4) = 4

- Come si ottiene un algoritmo?
  Si applica ripetutamente il procedimento fino a che non si ottiene che m=n.

Esempio:

m	n	maggiore - minore
210	63	147
147	63	84
84	63	21
21	63	42
21	42	21
21	21	

Algoritmo: di Euclide per il calcolo del MCD

```
int m,n;
scanf("%d%d", &m, &n);
while (m != n)
  if (m > n)
    m = m - n;
  else
    n = n - m;

printf("MCD: %d\n", m);
```

- Cosa succede se m=n=0?

  $\Longrightarrow$ il risultato è 0

- E se m=0 e n$\neq$ 0 (o viceversa)?

  $\Longrightarrow$ si entra in un ciclo infinito

- Per assicurarci che l'algoritmo venga eseguito su valori corretti, possiamo inserire una verifica sui dati in ingresso, attraverso un ciclo di lettura

  Proposte?

```
do {
  printf("Immettere due interi positivi:  ");
  scanf("%d%d", &m, &n);
  if (m <= 0 || n <= 0)
    printf("Errore:  i numeri devono essere > 0!\n");
} while (m <= 0 || n<= 0);
```

Metodo di Euclide con i resti per il calcolo del MCD

- Cosa succede se $m \gg n$?

Esempio:

MCD(1000, 2)

1000	2
998	2
996	2
...	
2	2

MCD(1001, 500)

1001	500
501	500
1	500
...	
1	1

- Come possiamo comprimere questa lunga sequenza di sottrazioni?
- Metodo di Euclide: sia

$$m = n \cdot k + r \quad (\text{con } 0 \leq r < m)$$

$$\begin{aligned} MCD(m, n) &= n && \text{se } r=0 \\ MCD(m, n) &= MCD(r, n) && \text{se } r \neq 0 \end{aligned}$$

Algoritmo di Euclide con i resti per il calcolo del MCD

```
leggi m ed n
while m ed n sono entrambi ≠ 0
{ sostituisci il maggiore tra m ed n con
  il resto della divisione del maggiore per il minore
}
stampa il numero tra i due che e' diverso da 0
```

Esercizio

Tradurre l'algoritmo in C

Cicli annidati

- Il corpo di un ciclo può contenere a sua volta un ciclo.

  Esempio: Stampa della tavola pitagorica.

  Algoritmo

```
for ogni riga tra 1 e 10
  { for ogni colonna tra 1 e 10
    stampa riga * colonna
  stampa un a capo }
```

- Traduzione in C

```
int riga, colonna;
const int Nmax = 10; /* indica il numero di righe e di
colonne */
for (riga = 1; riga <= Nmax; riga=riga+1) {
  for (colonna = 1; colonna <= Nmax; colonna=colonna+1)
    printf("%d ", riga * colonna);
  putchar('\n'); }
```

Digressione sulle costanti: la direttiva `#define`

- Nel programma precedente, `Nmax` è una costante. Tuttavia la dichiarazione

  ```
  const int Nmax = 10;
  ```

  causa l'allocazione di memoria (si tratta duna dichiarazione di variabile *read only*)
- C'è un altro modo per ottenere un identificatore costante, che utilizza la direttiva **#define**.

  ```
  #define Nmax 10
  ```
- **#define** è una direttiva di compilazione
- dice al compilatore di sostituire ogni occorrenza di `Nmax` con `10` prima di compilare il programma
- a differenza di **const** non alloca memoria

Assegnamento e altri operatori

- In C, l'operazione di assegnamento `x = exp` è un'espressione
  - il valore dell'espressione è il valore di `exp` (che è a sua volta un'espressione)
  - la valutazione dell'espressione `x = exp` ha un side-effect: quello di assegnare alla variabile `x` il valore di `exp`
- Dunque in realtà, "=" è un operatore (associativo a destra).
  Esempio: Qual'è l'effetto di `x = y = 4` ?
  - È equivalente a: `x = (y = 4)`
  - `y = 4` ... espressione di valore `4` con modifica (side-effect) di `y`
  - `x = (y = 4)` ... espressione di valore `4` con ulteriore modifica su `x`
- L'eccessivo uso di assegnamenti come espressioni rende il codice difficile da comprendere e quindi correggere/modificare.

Operatori di incremento e decremento

- Assegnamenti del tipo:
```
i = i + 1
i = i - 1
```
sono molto comuni.
  - operatore di incremento: `++`
  - operatore di decremento: `--`
- In realtà `++` corrisponde a due operatori:
- postincremento: `i++`
  - il valore dell'espressione è il valore di `i`
  - side-effect: incrementa `i` di `1`
- L'effetto di

```
int i,j;
i=6;
j=i++;
```

è j=6, i=7.

- preincremento: `++i`
  - il valore dell'espressione è il valore di `i+1`
  - side-effect: incrementa `i` di `1`
- L'effetto di

```
int i,j;
i=6;
j=++i;
```

è j=7, i=7.

(analogamente per `i--` e `--i`)

- Nota sull'uso degli operatori di incremento e decremento

Esempio:

	Istruzione	x	y	z
1	`int x, y, z;`	?	?	?
2	`x = 4;`	4	?	?
3	`y = 2;`	4	2	?
4a	`z = (x + 1) + y;`	4	2	7
4b	`z = (x++) + y;`	5	2	6
4c	`z = (++x) + y;`	5	2	7

- N.B.: Non usare mai in questo modo!
  In un'istruzione di assegnamento non ci devono essere altri side-effect (oltre a quello dell'operatore di assegnamento) !!!
- Riscrivere, ad esempio, come segue:

4b: `z = (x++) + y;` $\Longrightarrow$

```
z = x + y;
x++;
```

4c: `z = (++x) + y;` $\Longrightarrow$

```
x++;
z = x + y;
```

Ordine di valutazione degli operandi

- In generale il C non stabilisce quale è l'ordine di valutazione degli operandi nelle espressioni.

  Esempio:
  ```
  int x, y, z;
  x = 2;
  y = 4;
  z = x++ + (x * y);
  ```

- Quale è il valore di `z`?
  - se viene valutato prima `x++`: $2 + (3 * 4) = 14$
  - se viene valutato prima `x*y`: $(2 * 4) + 2 = 10$

Forme abbreviate dell'assegnamento

```
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;
```