UNITÀ DI APPRENDIMENTO **A3**
LE FUNZIONI

IN QUESTA UNITÀ IMPARERAI...

- Come scomporre in sottoprogrammi
- A conoscere i vari tipi di sottoprogrammi
- Come avviene il passaggio dei parametri
- A scrivere algoritmi che fanno uso di sottoprogrammi

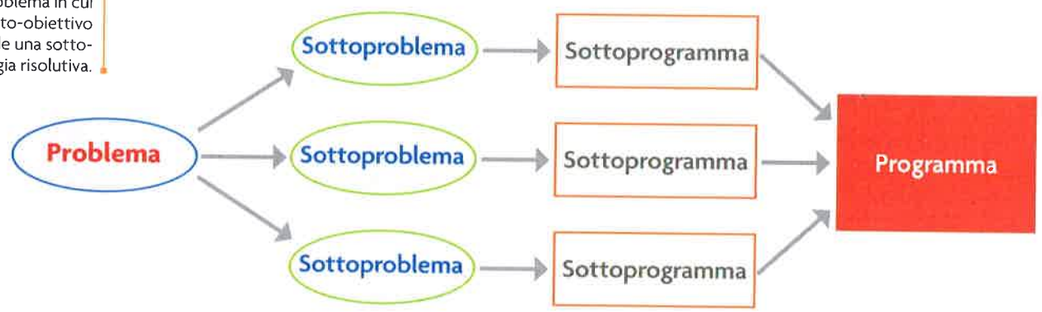
File sorgenti · Laboratorio "Case study" · Test · Glossario CLIL

1 Top-down e bottom-up

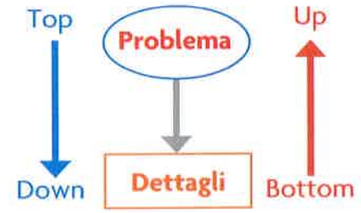
LO SAI CHE...

La tecnica **top-down** parte dall'obiettivo e da esso fa scaturire la strategia più adatta a raggiungerlo; valorizza il *perché* e da esso fa dipendere il *come*, ossia la strategia. Individua, pertanto, le risorse necessarie, precisa quelle disponibili e identifica quelle mancanti, propone successivamente ogni risorsa mancante come sotto-obiettivo, ovvero come sottoproblema in cui ciascun sotto-obiettivo richiede una sotto-strategia risolutiva.

Quando un problema appare complesso, per risolverlo possiamo individuare e analizzare i **sottoproblemi** più semplici che lo compongono, oltre alle loro interrelazioni. In questo modo è possibile articolare la progettazione dell'algoritmo complessivo in una serie di algoritmi più semplici, che verranno poi opportunamente assemblati.
Tale metodologia, di natura gerarchica, prende il nome di **top-down**, ossia "dall'alto verso il basso". Gli aggettivi alto e basso si riferiscono al **livello di dettaglio** o **astrazione**. Il livello più alto (*top*) è quello generale, chiamato **problema principale**; in esso si individuano i nodi fondamentali chiamati **sottoproblemi**. Ciascun sottoproblema viene dettagliato a parte e, se complesso, può essere a sua volta scomposto in ulteriori sottoproblemi più semplici. In sintesi, si scende dal generale al particolare mediante **affinamenti successivi**.
La tecnica top-down, quindi, nasce come **tecnica di analisi dei problemi** e non come tecnica di progettazione: il risolutore, infatti, utilizza tale metodologia per affrontare agevolmente il processo risolutivo del problema.
Per scomporre un problema in tanti sottoproblemi funzionali, ci si sofferma sul *cosa* debba essere fatto e non sul *come*, che con tale metodologia viene affrontato poco, e soltanto all'ultimo livello.



La metodologia **bottom-up**, ossia "dal basso verso l'alto", privilegia invece l'aspetto esecutivo rispetto a quello funzionale, procedendo dal particolare verso il generale.



Il metodo bottom-up è una **strategia induttiva** e consente di concentrarsi subito sui punti cardine del problema che, però, potrebbero essere di difficile individuazione iniziale. Nel modello top-down si affronta il problema osservandolo più in generale e poi rifinendo ogni sua parte. Ad esempio, creo un modello di automobile e poi scendo nel dettaglio rifinendo ruote, motore e così via. Un approccio informatico è quello di servirsi di sottoprogrammi da definire in un secondo momento: è il classico approccio della **programmazione procedurale**.
Nel modello bottom-up si affronta il problema preoccupandosi prima dei dettagli più semplici, fino ad arrivare al modello più complesso (il contrario del modello top-down). Ad esempio: creo una ruota, poi un motore e le altre parti e solo alla fine ottengo il modello complesso di un'automobile. Nella **programmazione a oggetti** è utilizzato questo metodo, in quanto si ha la possibilità di creare singoli oggetti indipendenti e quindi, tramite questo approccio, si sviluppa cominciando dalle classi base che poi si estendono o si collegano insieme a creare un programma complesso.
I moderni approcci alla progettazione software combinano spesso sia la tecnica top-down, sia quella bottom-up. Benché l'analisi e la comprensione del sistema completo siano tipicamente considerate necessarie per una buona progettazione (e quindi tramite l'approccio top-down), nella maggior parte dei progetti software si cerca di fare uso di codice già esistente ad alcuni livelli (tendenza bottom-up).

2 Sottoalgoritmi e sottoprogrammi

È possibile realizzare un sottoalgoritmo per ogni sottoproblema non più scomponibile. Alla fine, unendo tutti i sottoalgoritmi, si ottiene l'algoritmo che risolve il problema originale.

Il **sottoalgoritmo**, quindi, è una parte dell'algoritmo che risolve un particolare sottoproblema. Tecnicamente, è una parte dell'algoritmo risolutivo che non può essere eseguita autonomamente, ma soltanto su richiesta (invocazione) e sotto stretto controllo dell'algoritmo o del sottoalgoritmo che lo invoca, il quale per tale motivo, è denominato "chiamante".

È possibile suddividere il procedimento generale che risolve un problema in:

- un **algoritmo principale** (**main**) che descrive globalmente il problema;
- un insieme di **sottoalgoritmi** che risolvono i singoli sottoproblemi.

Analizziamo il seguente problema: *Preparare una torta gelato al cioccolato*. Realizziamo l'algoritmo servendoci dello pseudolinguaggio:

PSEUDOCODICE

```
ALGORITMO Torta
INIZIO
    Preparare la base per la torta
    Preparare un gelato al cioccolato
    Preparare la glassa al cioccolato
    Farcire la torta con il gelato e la glassa
FINE
```

Tutte e quattro le azioni sono piuttosto complesse e anche di interesse generale: per questi motivi conviene descriverle per mezzo di sottoprogrammi. Continuiamo l'affinamento, partendo dalla prima azione, cioè *Preparare la base per la torta*. Avremo:

PSEUDOCODICE

```
SOTTOALGORITMO Preparare la base per la torta
INIZIO
    Preparare l'impasto
    Preparare la teglia
    Preparare il forno
    Infornare
    Sfornare
FINE
```

Ogni algoritmo viene tradotto in un programma quando si utilizza un linguaggio di programmazione. I sottoalgoritmi, una volta codificati, vengono chiamati sottoprogrammi. Pertanto, si parla correttamente di programmi e sottoprogrammi quando si passa alla fase di "codifica" in un linguaggio di programmazione. Nella nostra trattazione, per uniformarci alla classica e consolidata terminologia in uso in ambiente informatico, utilizzeremo anche i termini programma principale e main per riferirci all'algoritmo principale e il termine sottoprogramma riferirci al sottoalgoritmo.

Applicare la tecnica top-down in modo eccessivo, ossia frammentare i vari sottoproblemi riducendoli a pochissime azioni ciascuno, è sconveniente. Il programma sarà costituito, in un caso del genere, da tantissimi sottoprogrammi (spesso inutili e talvolta contenenti una sola istruzione). Diverrà, così, eccessivamente frammentato, inefficiente e scarsamente leggibile.

L'attività *Preparare l'impasto* è di interesse generale, in quanto lo stesso impasto potrà essere utilizzato per molti altri tipi di torte: quindi si può descriverla, laddove fosse necessario, tramite un sottoprogramma. Le altre (da *Preparare la teglia* a *Sfornare*) sono piuttosto semplici e pertanto possono essere dettagliate immediatamente. Analogo ragionamento andrà fatto per le altre azioni descritte nell'algoritmo principale.
Uno stesso sottoprogramma può essere richiamato più volte sia dal programma principale, sia dagli altri sottoprogrammi. Inoltre, può essere utilizzato all'interno di altri programmi nei quali sia necessaria la stessa operazione; ciò consente la riusabilità del codice.

Conviene descrivere un'attività per mezzo di un sottoalgoritmo quando	Non conviene descrivere un'attività per mezzo di un sottoalgoritmo se
• è di interesse generale • non è di interesse generale ma si presenta più volte all'interno del programma; • non migliora la leggibilità del programma o, addirittura, la peggiora.	• è di scarso interesse generale • pur essendo di scarso interesse generale, permette una maggiore leggibilità del programma

Riepilogando, esistono ottimi motivi che spingono a utilizzare i sottoprogrammi. In particolare essi:

- **migliorano la leggibilità** del programma in maniera considerevole;
- **permettono l'astrazione funzionale**; quando il programmatore inserisce un'istruzione di chiamata di sottoprogramma, astrae dalla realtà del sottoproblema, cioè si disinteressa, in quel momento, della sua realizzazione, perché gli interessa solo che cosa fare e non come farlo;
- consentono di scrivere meno codice e così **occupano meno memoria**; si evita, infatti, di riscrivere più volte sequenze di istruzioni identiche in punti diversi del programma;
- **sono riutilizzabili**. Molto spesso accade che il sottoproblema da affrontare sia già stato risolto altrove; se disponiamo già del sottoalgoritmo risolutivo, possiamo riutilizzarlo senza riscriverlo. Per poter essere facilmente riutilizzabile, il sottoalgoritmo deve avere una forte coesione e pochi collegamenti con l'esterno; deve, cioè, risolvere il suo compito e avere pochi dati in comune con l'esterno.

La struttura generale di un algoritmo con sottoalgoritmi è la seguente:

PSEUDOCODICE

```
ALGORITMO NomeAlgoritmo
Parte dichiarativa globale          <-- Risorse dell'intero algoritmo
Main()
Parte dichiarativa del Main         <-- Risorse dell'algoritmo principale
INIZIO
  Corpo dell'algoritmo principale (Main)
FINE
SOTTOALGORITMO NomeSottoalgoritmo1
Parte dichiarativa di NomeSottoalgoritmo1   <-- Risorse di <NomeSottoalgoritmo1>
INIZIO
  Corpo di NomeSottoalgoritmo1
FINE
SOTTOPROGRAMMA NomeSottoalgoritmoN
Parte dichiarativa di NomeSottoalgoritmoN   <-- Risorse di <NomeSottoalgoritmo1>
INIZIO
  Corpo di <NomeSottoalgoritmoN
FINE
```


3 Tipologie di sottoprogrammi

Nella teoria informatica si è soliti distinguere due tipi di sottoprogrammi:

- **procedure**: sottoprogrammi che non restituiscono alcun valore: concernono una fase dell'elaborazione e non possono far parte di espressioni;
- **funzioni**: sottoprogrammi che restituiscono al programma chiamante un valore, assegnabile eventualmente a una variabile; le funzioni vengono utilizzate principalmente a destra del segno di assegnazione (=) e possono essere impiegate anche all'interno di espressioni.

Il nostro percorso prevede l'utilizzo del linguaggio C/C++, per cui è bene sapere che in questo linguaggio i sottoprogrammi si chiamano genericamente funzioni.

Più in dettaglio:

- **funzioni che non restituiscono valori** (le nostre procedure);
- **funzioni che restituiscono un valore** (le nostre funzioni).

4 Le procedure

La procedura è un sottoprogramma contenente le istruzioni che risolvono uno specifico problema. L'esecuzione di una procedura viene attivata dall'apposita istruzione di chiamata (invocazione).

Le procedure sono sempre associate a un nome simbolico, che viene indicato nella prima istruzione del sottoprogramma e in ogni istruzione di chiamata del medesimo.

In pseudocodifica, la sintassi generale di una procedura è la seguente:

```
PROCEDURA <Nome>([<ListaParametri>])
{
  <Istruzioni>
}
```

dove:

- <Nome> è il nome grazie al quale è possibile richiamare la procedura. Il nome va scritto tenendo presenti le regole utilizzate per gli identificatori delle variabili, ossia con la sola lettera iniziale maiuscola e, nel caso di nomi composti, con le lettere iniziali maiuscole (ad esempio *Ordina*, *TrovaMassimo*);
- <ListaParametri> rappresenta l'elenco degli eventuali parametri che vengono trasmessi alla procedura secondo la seguente sintassi:

```
(<Parametro1>: <TipoParametro1>; <Parametro2>: <TipoParametro2>; ...;
                                                  <ParametroN>: <TipoParametroN>)
```

- I parametri permettono lo scambio di informazioni in input e/o in output tra il programma chiamante e la procedura stessa. Nel caso in cui la procedura non contenga parametri, il suo nome deve comunque essere seguito da una parentesi tonda aperta e da una chiusa ().

Per il momento ci occupiamo di procedure senza liste di parametri, al fine di familiarizzare con questo nuovo strumento.

OSSERVA COME SI FA

1. Risolviamo il seguente problema che ci aiuterà a comprendere l'utilizzo delle procedure:

Scrivere un algoritmo che, dati in input due numeri interi A e B, li visualizzi in ordine crescente.

Considerata l'estrema semplicità del problema, tralasciamo la fase di analisi e passiamo direttamente alla formalizzazione dell'algoritmo. Per la stesura dello pseudocodice seguiremo l'andamento top-down, nel senso che per prima cosa definiremo il programma principale e poi le procedure.

PSEUDOCODICE

```
ALGORITMO Ordinamento
VARIABILI
  A, B, C: INTERO
Main()
INIZIO
  SCRIVI("Inserisci il primo numero")
  LEGGI(A)
  SCRIVI("Inserisci il secondo numero")
  LEGGI(B)
  ORDINA()  // Chiamata della procedura Ordina: il controllo passa
            // al sottoprogramma Ordina
  SCRIVI("I numeri ordinati sono ", A, B)
FINE

PROCEDURA Ordina()
INIZIO
  SE(A > B)
    ALLORA
      C ← A
      A ← B
      B ← C
  FINESE
FINE // Il controllo passa all'istruzione successiva a quella di chiamata
```

Analizziamo lo pseudocodice. All'interno del Main è presente l'istruzione:

```
Ordina()
```

che rappresenta la chiamata della procedura. Quando l'esecutore incontra questa istruzione, sospende l'esecuzione del Main e passa a eseguire il sottoprogramma corrispondente, ossia passa all'istruzione:

```
PROCEDURA Ordina()
```

Il sottoprogramma viene così eseguito. Quando l'esecutore incontra l'istruzione:

```
FINE
```

della procedura Ordina, ritorna a eseguire il Main partendo dall'istruzione successiva a quella della chiamata, ossia esegue l'istruzione:

```
SCRIVI("I numeri ordinati sono ", A, B)
```


5 Le funzioni void in C/C++

Di seguito è riportata la dichiarazione di una funzione void in pseudolinguaggio e in C++:

PSEUDOCODICE	C/C++
PROCEDURA <Nome>([<ElencoParametri>])	void <Nome>([<ElencoParametri>])
INIZIO	{
<Istruzioni>	<Istruzioni>
FINE	}

La parola chiave **void** viene utilizzata per indicare che la funzione non deve ritornare alcun valore al programma chiamante.
Nome rappresenta il nome assegnato alla funzione e segue le stesse regole sintattiche dei nomi da assegnare alle variabili. Le parentesi tonde racchiudono l'eventuale elenco dei parametri da passare alla funzione e le parentesi graffe delimitano il blocco che contiene le istruzioni.
Per poter essere eseguita, la funzione necessita di una chiamata da parte del main() o di qualsiasi altra funzione.
Al fine di garantire un preciso "ordine mentale", come abbiamo già detto nel paragrafo precedente, il nostro pseudocodice prevede che i sottoprogrammi siano riportati dopo l'algoritmo principale (main); nel caso del C/C++, invece, la dichiarazione dei sottoprogrammi è posta prima che essi vengano chiamati. Pertanto, se è il main a invocare una funzione, questa deve necessariamente essere inserita prima del main. L'utilizzo dei prototipi, che vedremo più avanti, consentirà di ovviare a tale problema.

OSSERVA COME SI FA

1. Scrivere un programma per il calcolo delle soluzioni di un'equazione di secondo grado; utilizzare le procedure per uno sviluppo top-down del problema.

Ricordiamo che, per risolvere un'equazione di secondo grado scritta nella forma canonica:

$$ax^2 + bx + c = 0$$

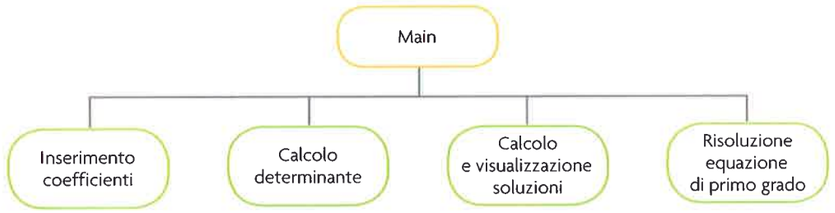
occorre conoscere il valore dei tre coefficienti *a*, *b* e *c*, dai quali le soluzioni dipendono.
Una volta acquisiti tali valori, occorre calcolare il discriminante (Δ) utilizzando la formula: $b^2 - 4 \times a \times c$.
In base al valore del discriminante, si verificano tre distinte situazioni:

- se $\Delta < 0$ non esistono soluzioni reali;
- se $\Delta = 0$ l'equazione ammette due soluzioni reali coincidenti;
- se $\Delta > 0$ l'equazione ammette due soluzioni reali distinte.

Nei casi in cui le soluzioni esistono, le radici reali si ottengono con la formula:

$$\frac{-b \pm \sqrt{\Delta}}{2a}$$

A questo punto, è necessario fare attenzione al valore assegnato al coefficiente *a*: infatti, se questo è uguale a zero, non solo non sarà possibile applicare la formula, ma ci si troverà di fronte a un'equazione non più di secondo grado, ma di primo grado.
L'analisi appena condotta ci porta a individuare quattro sottoprogrammi; il top-down è pertanto il seguente:



PSEUDOCODICE

```
ALGORITMO EquazSecondoGrado
VARIABILI
  A, B, C: INTERO
  Delta, X1, X2, X: REALE

Main()
INIZIO
  AcquisisciCoefficienti()
  SE(A ≠ 0)
    ALLORA
      CalcolaDelta()
      VisualizzaSoluzioni()
    ALTRIMENTI
      RisolviEquazPrimoGrado()
  FINESE
FINE

PROCEDURA AcquisisciCoefficienti()
INIZIO
  SCRIVI("Inserisci il valore del coefficiente a")
  LEGGI(A)
  SCRIVI("Inserisci il valore del coefficiente b")
  LEGGI(B)
  SCRIVI("Inserisci il valore del coefficiente c")
  LEGGI(C)
FINE

PROCEDURA CalcolaDelta()
INIZIO
  Delta ← B * B – 4 * A * C
FINE

PROCEDURA VisualizzaSoluzioni()
INIZIO
  SE(Delta < 0)
    ALLORA
      SCRIVI("L'equazione non ammette soluzioni reali")
    ALTRIMENTI
      X1 ← (–B – RADQUADRATA(Delta)) / (2 * A)
      X2 ← (–B + RADQUADRATA(Delta)) / (2 * A)
      SCRIVI("X1 = ", X1, "X2 = ", X2)
  FINESE
FINE

PROCEDURA RisolviEquazPrimoGrado()
INIZIO
  SE((B = 0) AND (C = 0))
    ALLORA
      SCRIVI("Equazione indeterminata")
    ALTRIMENTI
      SE(B = 0)
        ALLORA
          SCRIVI("Equazione impossibile")
        ALTRIMENTI
          X ← –C / B
      FINESE
  FINESE
FINE
```


C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int a, b, c;
float x1, x2, d;
void Acquisisci_Coefficienti()
{
  printf("Inserisci il valore del coefficiente a ");
  scanf("%d", &a);
  printf("\nInserisci il valore del coefficiente b ");
  scanf("%d", &b);
  printf("\nInserisci il valore del coefficiente c ");
  scanf("%d", &c);
}
void Calcola_Delta()
{
  d=(b*b)-(4*a*c);
}
void Visualizza_Soluzioni()
{
  if(d<0)
   printf("\nL'equazione non ammette soluzioni reali\n");
  else
  {
    x1 = (-b - sqrt(d))/(2*a);
    x2 = (-b + sqrt(d))/(2*a);
    printf("\nx1= %-3.1f x2= %-3.1f\n", x1, x2);
  }
}
void Risolvi_Equazione()
{
  if((b==0) && (c==0))
    printf("\nEquazione indeterminata\n");
  else
    if(b==0)
      printf("\nEquazione impossibile\n");
    else
      x1 = -c/b;
      printf("\nx= %-3.1f \n", x1);
}
main()
{
  Acquisisci_Coefficienti();
  if(a != 0)
  {
    Calcola_Delta();
    Visualizza_Soluzioni();
  }
  else
    Risolvi_Equazione();
  system("PAUSE");
}
```

La libreria math consente di utilizzare funzioni matematiche

C++

```
#include <math.h>
#include <iostream>
using namespace std;
int a, b, c;
float x1, x2, d;
void Acquisisci_Coefficienti()
{
  cout << "Inserisci il valore del coefficiente a ";
  cin >> a;
  cout << "\nInserisci il valore del coefficiente b ";
  cin >> b;
  cout << "\nInserisci il valore del coefficiente c ";
  cin >> c;
}
void Calcola_Delta()
{
  d=(b*b)-(4*a*c);
}
void Visualizza_Soluzioni()
{
  if(d<0)
   cout << "L'equazione non ammette soluzioni reali" << endl;
  else
  {
    x1 = (-b - sqrt(d))/(2*a);
    x2 = (-b + sqrt(d))/(2*a);
    cout << "x1=" << x1 << " x2=" << x2 << endl;
  }
}
void Risolvi_Equazione()
{
  if((b==0) && (c==0))
    cout << "\nEquazione indeterminata" << endl;
  else
    if(b==0)
      cout << "\nEquazione impossibile" << endl;
    else
      x1 = -c/b;
      cout << "x=" << x1 << endl;
}
main()
{
  Acquisisci_Coefficienti();
  if(a != 0)
  {
    Calcola_Delta();
    Visualizza_Soluzioni();
  }
  else
    Risolvi_Equazione();
  system("PAUSE");
}
```

```
Inserisci il valore del coefficiente a 2

Inserisci il valore del coefficiente b 3

Inserisci il valore del coefficiente c 1
x1=-1 x2=-0.5
Premere un tasto per continuare . . .
```

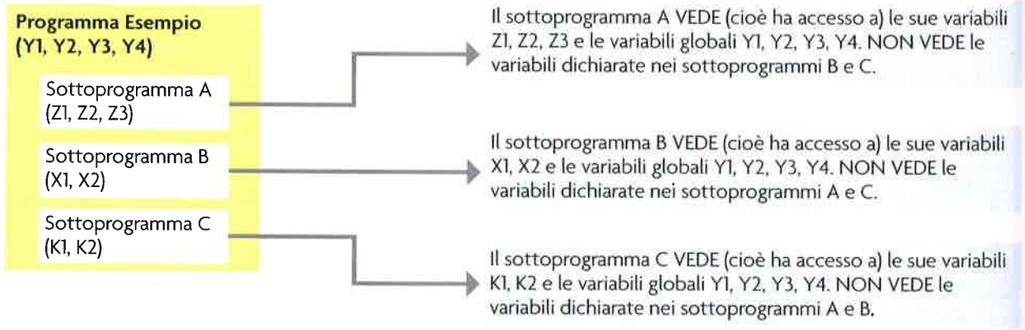
6 | Ambiente locale e ambiente globale

Durante l'implementazione di un sottoprogramma occorre definire tutte le risorse necessarie al suo funzionamento, vale a dire il suo **ambiente**.

Con il termine ambiente di un sottoprogramma definiamo l'insieme delle risorse (variabili, costanti, sottoprogrammi, parametri) alle quali esso può accedere.

Per il momento, diciamo che l'ambiente è costituito:

- dall'**ambiente locale**, ossia dalle risorse che sono dichiarate all'interno del sottoprogramma (**risorse locali**);
- dall'**ambiente globale**, ossia dalle risorse che sono utilizzabili da tutti i sottoprogrammi (**risorse globali**).



Consideriamo l'*algoritmo che, dati in input due numeri interi A e B, li visualizza in ordine crescente.*

PSEUDOCODICE

```
ALGORITMO Ordinamento
VARIABILI
  A, B : INTERO
Main()
INIZIO
  SCRIVI("Inserisci il primo numero")
  LEGGI(A)
  SCRIVI("Inserisci il secondo numero")
  LEGGI(B)
  Ordina()  // Chiamata della procedura Ordina: il controllo
            // passa al sottoprogramma Ordina
  SCRIVI("I numeri ordinati sono ", A, B)
FINE
PROCEDURA Ordina()
VARIABILI
  C : INTERO
INIZIO
  SE(A > B)
    ALLORA
      C ← A
      A ← B
      B ← C
  FINESE
FINE      // Il controllo passa all'istruzione successiva
          // a quella di chiamata
```

C++

```
#include <iostream>
using namespace std;
int A,B;
void Ordina()
{
  int C;
  if(A>B)
    {
      C=A;
      A=B;
      B=C;
    }
}
main()
{
  cout <<"inserisci il primo numero ";
  cin >> A;
  cout <<"inserisci il secondo numero ";
  cin >> B;
  Ordina();
  cout << "i numeri ordinati sono:" << A << " e " << B << endl;
  system("PAUSE");
}
```


A e *B* sono **variabili globali** e possono essere utilizzate da tutti i sottoprogrammi dichiarati all'interno dell'algoritmo.
La variabile *C*, invece, è dichiarata all'interno del sottoprogramma e può essere utilizzata solo da esso. Viene quindi utilizzata solo durante l'esecuzione del sottoprogramma: è, pertanto, una **variabile locale**. La scelta delle variabili locali non deve essere affidata al caso. Nel nostro esempio, infatti, abbiamo deciso di definire *C* come variabile locale perché viene utilizzata esclusivamente dal sottoprogramma. L'utilizzo delle variabili locali permette di:

- agevolare la lettura del programma, in quanto mette in evidenza in quale ambito hanno significato le risorse;
- individuare facilmente eventuali errori commessi, in quanto ci si sofferma solo sulle risorse locali nell'ambito di quel sottoprogramma.

Le variabili globali	Le variabili locali
sono allocate (e inizializzate – in ogni caso)	sono allocate (e inizializzate, se richiesto) quando si entra nel sottoprogramma
rimangono allocate per tutta la durata del sottoprogramma	rimangono allocate solo per la durata del programma
vengono inizializzate solo una volta	vengono inizializzate tutte le volte che viene eseguito il sottoprogramma

Un corretto stile di programmazione impone di minimizzare l'uso dell'ambiente globale e di privilegiare quello locale. L'uso di variabili globali è fortemente sconsigliato e comunque richiede parsimonia, poiché rischia di generare interazioni di difficile controllo tra diverse parti di un programma. L'abuso di variabili globali può rendere difficile isolare gli errori (in gergo *bug*) e inoltre indica che il progetto di un programma non è stato pensato attentamente.

7 | Le regole di visibilità

All'interno di un programma ogni oggetto ha un suo **campo di validità** (**scope**), ossia un ambito in cui può essere usato e riconosciuto. È pertanto necessario definire delle regole per determinare il campo di visibilità degli oggetti globali e locali di un programma. Si parte dai seguenti principi:

1. Gli oggetti globali sono accessibili a (visibili in) tutto il programma.
2. Un oggetto dichiarato in un sottoprogramma ha significato solo in quel sottoprogramma e in tutti quelli dichiarati al suo interno. L'ambiente di un sottoprogramma, quindi, include anche tutte le risorse dei sottoprogrammi che contengono il sottoprogramma stesso.
3. Un oggetto non può essere usato se non è stato prima dichiarato

Nella descrizione di un algoritmo, può succedere che una variabile sia dichiarata con lo stesso nome di un'altra (il tipo potrebbe anche non essere uguale) a livello globale e a livello locale all'interno di un sottoprogramma. Si tratta di due variabili diverse che occupano diversi spazi di memoria, anche se hanno uguale nome. Osserva il seguente esempio:

PSEUDOCODICE

```
ALGORITMO Indovina
VARIABILI
  A : INTERO
Main()
INIZIO
  ...
FINE
PROCEDURA Sorpresa()
VARIABILI
  A : INTERO
INIZIO
  A ← INTERO
FINE
```

Quale variabile *A* assumerà valore 3? Quella globale o quella locale al sottoprogramma Sorpresa? Per rispondere al quesito si applica la regola di sovrapposizione e il concetto di **shadowing**, secondo il quale la variabile locale, durante l'esecuzione del sottoprogramma che la contiene, "oscura" (maschera) l'omonima variabile più esterna, impedendone la visibilità. La risposta è ora chiara: il valore 3 sarà assegnato alla variabile *A* locale mentre la *A* globale non verrà toccata.

ORA TOCCA A TE!

1 Stabilisci quali sono variabili globali e quali locali. Definisci, inoltre, che cosa "vede" e che cosa "non vede" ogni sottoprogramma.

```
PSEUDOCODICE
ALGORITMO X
VARIABILI
  A, B : INTERO
Main()
INIZIO
  ..............
FINE

PROCEDURA Uno()
VARIABILI
  C : INTERO
INIZIO
  ..............
FINE

PROCEDURA Due()
VARIABILI
  D : REALE
INIZIO
  ..............
FINE
```

2 Osserva il seguente pseudocodice; a quale variabile è assegnato il valore 2?

```
PSEUDOCODICE
ALGORITMO Indovina
VARIABILI
  X: REALE
Main()
INIZIO
  X ← 9
FINE

PROCEDURA Moltiplica()
VARIABILI
  X: REALE
INIZIO
  X ← 2
FINE
```


3 Traduci il seguente algoritmo in linguaggio C o C++.

```
PSEUDOCODICE
ALGORITMO Indovina
VARIABILI
  X: INTERO

Main()
INIZIO
  X ← 9
  Moltiplica()
FINE

PROCEDURA Moltiplica()
VARIABILI
  X: REALE
INIZIO
  X ← 2
  Somma()
FINE

PROCEDURA Somma()
VARIABILI
  Voc: INTERO
  X, Y, Z: REALE
INIZIO
  X ← 2
FINE
```

8 I parametri

Un sottoprogramma è più utile se è **funzionalmente indipendente** dal programma principale. Infatti, un sottoprogramma può essere utilizzato più volte all'interno di un programma e può anche essere trasportato, cioè utilizzato con successo in altri programmi.
Analizziamo il seguente pseudocodice, che visualizza un messaggio di saluto per tre differenti nominativi forniti in input:

```
PSEUDOCODICE
ALGORITMO VisualizzaNomi

Main()
INIZIO
  Visualizza1()
  Visualizza2()
  Visualizza3()
FINE

PROCEDURA Visualizza1()
INIZIO
  SCRIVI("Ciao" , "Mario")
FINE
```

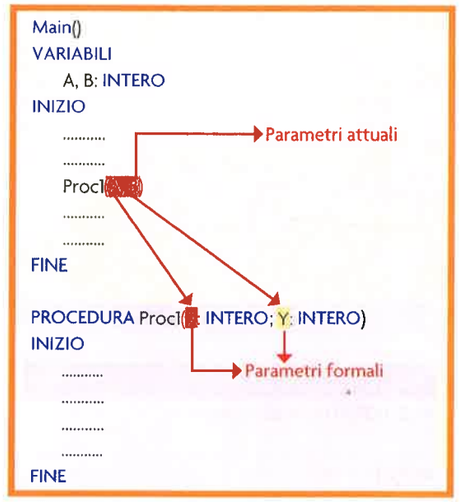
```
PROCEDURA Visualizza2()
INIZIO
  SCRIVI("Ciao" , "Paolo")
FINE

PROCEDURA Visualizza3()
INIZIO
  SCRIVI("Ciao" , "Fabio")
FINE
```

Le tre procedure sono identiche, ma operano su dati diversi; per questo motivo siamo stati costretti a riscrivere più volte il sottoprogramma, cambiando soltanto il suo nome e i dati su cui opera. Si rende necessario uno strumento che renda i **sottoprogrammi autonomi e indipendenti** dai dati del programma principale.
A questo scopo, i linguaggi di programmazione mettono a disposizione i **parametri**.

I **parametri** sono oggetti caratterizzati da:
- un identificatore;
- un tipo;
- un valore;
- una posizione;
- una direzione (input/output) che dipende dalla modalità di passaggio del parametro.

Grazie a essi si stabilisce come debbano avvenire l'input dei dati (al sottoprogramma) e l'output dei risultati (al sottoprogramma chiamante).

L'identificatore e il tipo dei parametri sono noti al momento della dichiarazione del sottoprogramma, ma il valore è noto solo all'atto della chiamata. I parametri permettono, quindi, di gestire la **comunicazione del sottoprogramma** con l'esterno. All'atto della chiamata del sottoprogramma occorre specificare i **parametri attuali**, ossia le informazioni reali che occorre trasmettergli. I valori di tali parametri saranno accolti dal sottoprogramma per mezzo dei **parametri formali** dichiarati nell'intestazione.
Per i parametri attuali occorre indicare solo il nome; per i parametri formali, invece, è necessario indicare il nome e il tipo.



È importante tenere presente che il **numero**, il **tipo** e l'**ordine** dei parametri attuali devono essere sempre uguali a quelli dei corrispondenti parametri formali. Nel nostro esempio, infatti, la chiamata del sottoprogramma Proc1 associa il parametro attuale A di tipo intero al parametro formale X (anch'esso, ovviamente, di tipo intero) e il parametro attuale B di tipo intero al parametro formale Y di tipo intero.

LO SAI CHE...

I parametri attuali e i parametri formali possono anche avere casualmente lo stesso nome, ma si consiglia di utilizzare nomi diversi per evitare inutili confusioni.

Spesso, all'atto della dichiarazione di un sottoprogramma, nasce il problema della scelta dei parametri. Suggeriamo di rispettare le seguenti regole:

1. identificare i dati essenziali e provenienti dall'esterno di cui il sottoprogramma ha bisogno;
2. identificare l'output che il programma chiamante vuole ottenere dal sottoprogramma.

A conclusione di questo paragrafo, vediamo come può essere trasformato il nostro algoritmo:

PSEUDOCODICE

```
ALGORITMO VisualizzaNomi
Main()
VARIABILI
  A, B, C: STRINGA
INIZIO
  A="Mario"
  B="Paolo"
  C="Fabio"
  Visualizza(A)
  Visualizza(B)
  Visualizza(C)
FINE
PROCEDURA Visualizza(S:STRINGA)
INIZIO
  SCRIVI("Ciao", S)
FINE
```

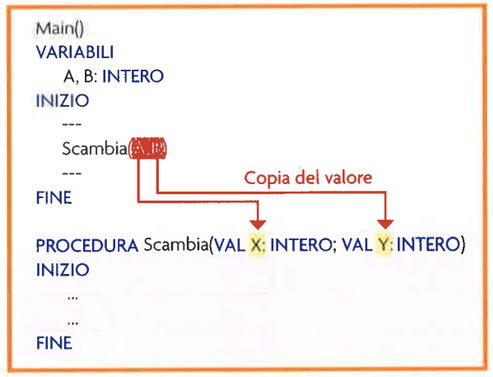
9 Il passaggio dei parametri per valore

Con **passaggio** o **trasmissione dei parametri** intendiamo l'operazione con la quale il valore dei parametri attuali viene associato (trasmesso) a quello dei parametri formali.

Il passaggio dei parametri può avvenire secondo due distinte modalità, ognuna rispondente a esigenze diverse:

- passaggio per valore o per copia (**by value**);
- passaggio per indirizzo o referenza (**by reference**).

Esaminiamo in dettaglio queste due modalità, cominciando dal passaggio per valore. Consideriamo la procedura Scambia() che dovrà scambiare i valori dei due parametri attivati.
Nel passaggio dei parametri per valore (*by value*) si ha soltanto una copia dei valori dei parametri attuali nei rispettivi parametri formali. Durante l'esecuzione del sottoprogramma, qualsiasi modifica apportata ai parametri formali sarà visibile solo all'interno del sottoprogramma e non verrà riportata sui parametri attuali (che continueranno, così, a conservare il valore iniziale).
I parametri formali vengono considerati come variabili locali il cui valore, al ritorno dal sottoprogramma, si perde.



Nel passaggio dei parametri per valore, all'atto della chiamata del sottoprogramma viene allocata un'area di memoria utilizzata per contenere i parametri formali. Si ha, così, una **duplicazione** dei dati relativi ai parametri. Al passaggio, i parametri formali verranno inizializzati con il valore dei rispettivi parametri attuali. In questo modo, il processore opera su questa nuova area di memoria *lasciando inalterati i parametri attuali*. Al rientro dal sottoprogramma, quest'area viene rilasciata, proprio come avviene per le variabili locali (a cui i parametri formali sono assimilabili.
Secondo il nostro formalismo, il passaggio per indirizzo viene indicato anteponendo la parola chiave VAL (*valore*) al parametro o alla lista di parametri formali interessati.

Servendoci dell'algoritmo per l'ordinamento di due numeri, vediamo graficamente come avviene questo passaggio.

PSEUDOCODICE

```
ALGORITMO OrdinaValori1
Main()
VARIABILI
   A, B: INTERO    // Variabili locali del Main
INIZIO
1.    SCRIVI("Inserisci il primo numero")
2.    LEGGI(A)
3.    SCRIVI("Inserisci il secondo numero")
4.    LEGGI(B)
5.    Scambia(A, B)
6.    SCRIVI("I numeri ordinati sono ", A, B)
FINE

PROCEDURA Scambia(VAL X: INTERO; VAL Y: INTERO)
VARIABILI
Comodo: INTERO    // Variabile locale della procedura Scambia
INIZIO
7.    SE(X > Y)
8.       ALLORA
9.          Comodo ← X
10.         X ← Y
11.         Y ← Comodo
12.   FINESE
13. FINE
```

Istruzione	Situazione di memoria
1	
2	4 A ☐ B
3	
4	4 A 1 B
5	4 A 1 B 4 X 1 Y ☐ Comodo
6	
7	
8	
9	4 A 1 B 4 X 1 Y 4 Comodo
10	4 A 1 B 1 X 1 Y 4 Comodo
11	4 A 1 B 1 X 4 Y 4 Comodo
12	
13	
6	4 A 1 B

È evidente che il programma, pur essendo corretto in ogni suo punto, non risolve il problema dato: le variabili A e B, infatti, non hanno subito il dovuto scambio. Il motivo per il quale l'algoritmo precedente non produce il risultato corretto è dato dal fatto che è stato utilizzato un passaggio di parametri per valore. Per questo tipo di problemi, quando cioè vogliamo che il sottoprogramma restituisca il **valore dei parametri formali** (quindi alteri il valore dei parametri attuali), dobbiamo servirci di un passaggio per indirizzo.

ORA TOCCA A TE!

1 Qual è il valore assunto dai parametri attuali dopo l'esecuzione del seguente pseudocodice? Qual è l'output fornito?

PSEUDOCODICE

```
ALGORITMO VisualizzaVariabili
Main()
VARIABILI
   X, K, W: INTERO
INIZIO
   SCRIVI("Inserisci il valore di X")
   LEGGI(X)
   SCRIVI("Inserisci il valore di K")
   LEGGI(K)
   SCRIVI("Inserisci il valore di W")
   LEGGI(W)
   Sorpresa(X,K,W)
   SCRIVI("La procedura ha restituito i seguenti risultati X=",X," K=",K," W=",W);
FINE
PROCEDURA Sorpresa(VAL G: INTERO; VAL H: INTERO; VAL J: INTERO)
INIZIO
   G ← 2
   H ← 3
   J ← 4
FINE
```

2 Trova gli errori presenti nel seguente pseudocodice.

PSEUDOCODICE

```
Main()
VARIABILI
   X, K, W: INTERO
INIZIO
   SCRIVI("Inserisci il valore di X ")
   LEGGI(X)
   SCRIVI("Inserisci il valore di K ")
   LEGGI(K)
   SCRIVI("Inserisci il valore di W ")
   LEGGI(W)
   Sorpresa(X,K,W)
   SCRIVI("La procedura ha restituito i seguenti risultati X=",X," K=",K," W=",W)
FINE
PROCEDURA Sorpresa(VAL G: CARATTERE; VAL H: CARATTERE; VAL J: CARATTERE)
INIZIO
   G ← 'G'
   H ← 'H'
   J ← 'J'
FINE
```

3 Qual è il valore assunto dalla variabile globale X al termine del seguente algoritmo?

PSEUDOCODICE

```
ALGORITMO IndovinaX
Main()
VARIABILI
  X, Y, K: INTERO
INIZIO
  X ← 2
  Y ← 4
  K ← 2
  Indovina(K)
FINE

PROCEDURA Indovina(VAL W: INTERO)
VARIABILI
  X, I: INTERO
INIZIO
  X ← 80
  PER I ← 1 A W ESEGUI
    X ← X + 10
  FINEPER
FINE
```

- 65,6
- 100
- 2
- 95,6
- Nessuno dei precedenti

10 Il passaggio dei parametri per valore in C/C++

In linguaggio C/C++ il passaggio dei parametri per valore non differisce da quanto abbiamo visto in pseudolinguaggio.
La sintassi generale è la seguente:

```
void <NomeFunzione>(<TipoDato1><Parametro1>,...,<TipoDatoN><ParametroN>)
{
  <Istruzioni>
}

main()
{
  <NomeFunzione>(Parametro1,...,ParametroN)
}
```


OSSERVA COME SI FA

1. Calcolare il valore della potenza a^x con a e x immessi in input.

C

```
#include <stdio.h>
#include <stdlib.h>
void Potenza(int b, unsigned int e)
{
  long int p;
  p=1;
  while(e>=1)
  {
    p *= b;
    e--;
  }
  printf("La potenza e\' %d\n",p);
}
main( )
{
  int a;
  unsigned int x;
  printf("Inserire la base ");
  scanf("%d", &a);
  printf("Inserire l'esponente ");
  scanf("%d", &x);
  Potenza(a,x);
  system("PAUSE");
}
```

C++

```
#include <iostream>
using namespace std;
void Potenza(int b, unsigned int e)
{
  long int p;
  p=1;
  while(e>=1)
  {
    p *= b;
    e--;
  }
  cout << "La potenza e\' " << p << endl;
}
main( )
{
  int a;
  unsigned int x;
  cout << "Inserire la base ";
  cin >> a;
  cout << "Inserire esponente ";
  cin >> x;
  Potenza(a,x);
  system("PAUSE");
}
```

2. Realizzare delle funzioni void per disegnare su video quadrati e triangoli di asterischi con valori inseriti in input.

C

```
#include <stdio.h>
#include <stdlib.h>
void StampaRiga(int lunghezza, char carattere)
{
  int i;
  for(i = 1 ; i <= lunghezza ; i++) printf("%c",carattere);
}
void StampaQuadrato(int altezza)
{
  int i, j; /* i e j sono variabili locali */
  for(i = 1; i <= altezza; i++)
  {
    StampaRiga(altezza,'*');
```

C++

```
#include <iostream>
using namespace std;
void StampaRiga(int lunghezza, char carattere)
{
  int i;
  for(i = 1 ; i <= lunghezza ; i++) cout << carattere;
}
void StampaQuadrato(int altezza)
{
  int i, j; /* i e j sono variabili locali */
  for(i = 1; i <= altezza; i++)
  {
    StampaRiga(altezza,'*');
```

```c
    printf("\n");
  }
}
void StampaTriangolo(int altezza)
{
  int i, j; /* i e j sono variabili locali!!! */
  for(i = 1; i <= altezza; i++) {
    StampaRiga(altezza – i,' ');
    StampaRiga(2*i – 1,'*');
    printf("\n"); }
}
main()
{
  char ch; int altezza;
  do
  {
    printf("\n digita \n");
    printf(" q: per stampare un quadrato\n");
    printf(" t: per stampare un triangolo\n");
    printf(" f: per terminare il programma:\n");
    scanf("%c", &ch);
    getchar(); /* serve per saltare il carattere '\n' */
    printf("\n");
    if (ch == 'q' || ch =='t')
    {
      printf("altezza?");
      scanf("%d",&altezza);
    }
    switch(ch)
    {
      case 'q' : StampaQuadrato(altezza); break;
      case 't' : StampaTriangolo(altezza); break;
    }
  } while (ch != 'f');
  system("PAUSE");
}
```

```c++
    cout << endl;
  }
}
void StampaTriangolo(int altezza)
{
  int i, j; /* i e j sono variabili locali!!! */
  for(i = 1; i <= altezza; i++) {
    StampaRiga(altezza - i,' ');
    StampaRiga(2*i - 1,'*');
    cout << endl; }
}
main()
{
  char ch; int altezza;
  do
  {
    cout << " digita " << endl;
    cout << " q: per stampare un quadrato" << endl;
    cout << " t: per stampare un triangolo" << endl;
    cout << " f: per terminare il programma:" << endl;
    cin >> ch;
    //getchar(); /* serve per saltare il carattere '\n' */
    cout << "\n";
    if (ch == 'q' || ch =='t')
    {
      cout << "altezza?";
      cin >> altezza;
    }
    switch(ch)
    {
      case 'q' : StampaQuadrato(altezza); break;
      case 't' : StampaTriangolo(altezza); break;
    }
  } while (ch != 'f');
  system("PAUSE");
}
```

```
 digita
 q: per stampare un quadrato
 t: per stampare un triangolo
 f: per terminare il programma:
t

altezza? 8
       *
      ***
     *****
    *******
   *********
  ***********
 *************
***************
```


ORA TOCCA A TE!

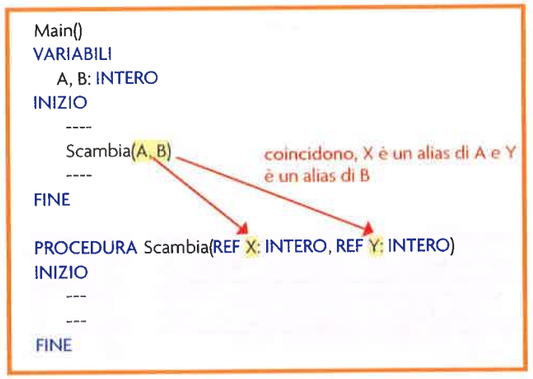
1 Modifica il seguente programma in modo da realizzare una funzione che riceve un carattere minuscolo/maiuscolo dell'alfabeto e lo converte in maiuscolo/minuscolo.

C

```c
#include <stdio.h>
#include <stdlib.h>
char carat;
main( )
{
  printf("Inserisci un carattere ");
  carat = getchar();    /* equivale a scanf("%c", &carat); */
  if(carat >= 'a' && carat <= 'z')
  {
    printf("Il carattere e\' in minuscolo\n");
    printf("Il maiuscolo e\' %c\n ", (char) ('A' + carat – 'a'));
  }
  else if(carat >= 'A' && carat <= 'Z')
  {
    printf("Il carattere e\' in maiuscolo\n");
    printf("Il minuscolo e\' %c\n", (char) ('a' + carat – 'A'));
  }
  else
  printf("Non hai inserito un carattere valido ");
  system("PAUSE");
}
```

11 Il passaggio dei parametri per indirizzo

Nel passaggio dei parametri per indirizzo (**by reference**) i parametri formali contengono l'indirizzo di memoria dei parametri attuali. Di conseguenza, una modifica dei parametri formali **provoca una modifica** dei corrispondenti attuali.



Con questo tipo di passaggio, quindi, si può perdere il contenuto originale dei parametri attuali. Secondo il nostro formalismo, il passaggio per indirizzo viene indicato anteponendo la parola chiave **REF** (*referenza*) al parametro o alla lista di parametri formali interessati.
Tecnicamente, il passaggio per indirizzo differisce da quello per valore in quanto, all'atto della chiamata del sottoprogramma, *non viene allocata alcuna area di memoria per valori dei parametri formali*. Poiché essi si riferiscono alla stessa area di memoria allocata per i parametri attuali, la stessa cella di memoria sarà individuabile con due nomi distinti (alias). In questo tipo di passaggio non viene trasmesso il valore dei parametri attuali, bensì *l'indirizzo della cella di memoria a essi assegnata*; di conseguenza, la modifica di un parametro comporterà la modifica dell'altro.
Consideriamo l'esempio dello scambio di variabili già analizzato durante la trattazione del passaggio per valore, ma questa volta effettuando un passaggio per indirizzo. La tabella riportata dopo lo pseudocodice illustra la situazione della memoria durante l'esecuzione.

Riportiamo di seguito alcune utili regole per una corretta progettazione di procedure con parametri.

- Le procedure devono comunicare con l'esterno esclusivamente tramite parametri.
- Non si deve utilizzare un numero eccessivo di parametri (se ciò dovesse accadere, sarebbe utile rivedere la progettazione).
- Le procedure non devono utilizzare le variabili globali e, in particolar modo, non devono modificarle (in altri termini, non devono produrre effetti collaterali o "side effect").
- Le procedure che risolvono un certo problema non dovrebbero usare al loro interno istruzioni di input/output: devono comunicare i dati solo attraverso i parametri.
- La gestione dell'input/output deve avvenire tramite apposite procedure.

PSEUDOCODICE

```
ALGORITMO OrdinaValori2
Main()
VARIABILI
   A, B: INTERO
INIZIO
1.    SCRIVI("Inserisci il primo numero")
2.    LEGGI(A)
3.    SCRIVI("Inserisci il secondo numero")
4.    LEGGI(B)
5.    Scambia(A,B)
6.    SCRIVI("I numeri ordinati sono ", A, B)
FINE
7.  PROCEDURA Scambia(REF X: INTERO; REF Y: INTERO)
VARIABILI
8.    Comodo: INTERO
INIZIO
9.    SE(X > Y)
10.      ALLORA
11.         Comodo ← X
12.         X ← Y
13.         Y ← Comodo
14.   FINESE
15. FINE
```

Istruzione	SItuazione di memoria
1	
2	4 A ☐ B
3	
4	4 A 1 B
5	4 A 1 B
7	4 A X 1 B Y
8	☐ Comodo
11	4 A X 1 B Y 4 Comodo
12	1 A X 1 B Y 4 Comodo
13	4 A X 4 B Y 4 Comodo
14	
15	4 A 1 B
6	4 A 1 B

Ora il risultato è quello che attendevamo!

OSSERVA COME SI FA

1. Scrivere un algoritmo che calcoli la potenza intera di un numero.

Dobbiamo innanzitutto chiedere in input il valore della base e quello dell'esponente, e accertarci che siano positivi, impedendo di accettare valori che non siano positivi. Per realizzare l'elevamento a potenza è sufficiente moltiplicare la base per se stessa per un numero di volte pari all'esponente. È evidente che il problema può essere scisso nei seguenti tre sottoproblemi:

PSEUDOCODICE

```
ALGORITMO CalcoloPotenza
Main()
VARIABILI
   Base, Esponente, Potenza: INTERO
INIZIO
   InserimentoDati(Base, Esponente)
   Potenza ← 1
   Eleva(Base, Esponente, Potenza)
   SCRIVI("La potenza calcolata è ", Potenza)
FINE
PROCEDURA InserimentoDati(REF A: INTERO; REF B: INTERO)
// I parametri A e B sono di output
INIZIO
   SCRIVI("Inserisci la base")
   LEGGI(A)
   RIPETI
      SCRIVI("Inserisci l'esponente")
      LEGGI(B)
   FINCHÉ(B > 0)
FINE
PROCEDURA Eleva(VAL A: INTERO; VAL B: INTERO; REF P: INTERO)
// I parametri A e B sono di input, P è di output
VARIABILI
   I: INTERO
INIZIO
   P ← 1
   PER I ← 1 A B ESEGUI
      P ← P * A
   FINEPER
FINE
```

ORA TOCCA A TE!

1 Occorre inviare i parametri A e B per indirizzo e il parametro C per valore. Quale tra le seguenti è l'intestazione corretta della procedura relativa alla chiamata?

```
VARIABILI
  A, B: INTERO
  C: REALE
INIZIO
  ......
  Modifica(A, B, C)
  ......
```

- **PROCEDURA** Modifica(A: **INTERO**; B: **INTERO**; C: **INTERO**)
- **PROCEDURA** Modifica(**REF** A: **INTERO**; **REF** B: **INTERO**; **VAL** C: **INTERO**)
- **PROCEDURA** Modifica(**REF** A: **INTERO**; **REF** B: **INTERO**; **VAL** C: **REALE**)
- **PROCEDURA** Modifica(**REF** A: **INTERO**; **REF** B: **INTERO**; **VAL** C: **INTERO**)

2 Qual è il valore assunto dai parametri attuali dopo l'esecuzione del seguente pseudocodice? Qual è l'output fornito?

PSEUDOCODICE

```
ALGORITMO Uno
Main()
VARIABILI
  X, K, W, Y: INTERO
INIZIO
  SCRIVI("Inserisci il valore di X ")
  LEGGI(X)
  SCRIVI("Inserisci il valore di K ")
  LEGGI(K)
  SCRIVI("Inserisci il valore di W ")
  LEGGI(W)
  Sorpresa(X, K, W)
  SCRIVI("La procedura ha restituito i seguenti risultati X=",X," K=",K," W=",W)
FINE
PROCEDURA Sorpresa (REF G: INTERO; REF H: INTERO; REF J: INTERO)
INIZIO
  G ← 2
  H ← 3
  J ← 4
FINE
```


3 Trova gli errori presenti nel seguente pseudocodice.

PSEUDOCODICE

```
ALGORITMO TrovaErrori
Main()
VARIABILI
  A, B, X: INTERO
INIZIO
  SCRIVI("Inserisci un numero")
  LEGGI(A)
  SCRIVI("Inserisci un altro numero")
  LEGGI(B)
  X ← Alterna(A,B)
  SCRIVI("I numeri ordinati sono ", A, B)
FINE

PROCEDURA Scambia(REV X, Y)
INIZIO
  SE(X > Y)
    ALLORA
      Comodo = X
      X = Y
      Y = Comodo
  FINESE
FINE
```

12 Il passaggio dei parametri per indirizzo in C/C++

Abbiamo visto che, in questo tipo di passaggio, non viene trasmesso il valore dei parametri attuali, bensì l'**indirizzo della cella di memoria a essi assegnata**. Il passaggio di parametri in C avviene solo per valore. Il passaggio per indirizzo è ottenuto passando l'indirizzo di memoria della variabile. Per indicare questo particolare in linguaggio C ci si serve di due operatori:

- **&**, che abbiamo utilizzato nella funzione scanf, chiamato **operatore di indirizzo**;
- *, chiamato **operatore di indirezione**.

Vediamo come si utilizzano questi due operatori. Osserva la seguente sintassi:

```
void <NomeProcedura>(<TipoDato1>*<Parametro1>,...,<TipoDatoN>*<ParametroN>)
{
  <Istruzioni>
}
main()
{
  <NomeProcedura>(&Parametro1,...,&ParametroN)
}
```

L'operatore *, applicato al nome del parametro attuale, fa riferimento al suo contenuto

L'operatore &, applicato al nome del parametro formale, trasmette l'indirizzo della cella di memoria e non il suo contenuto

Il C++, pur mantenendo questa modalità, ne presenta una sintatticamente più semplice: il parametro che deve essere passato per riferimento è preceduto, nell'intestazione della funzione, dall'operatore &. Tutto qui.

LO SAI CHE...

Il tipo **void**, come abbiamo già visto in qualche esempio, può essere utilizzato anche per indicare l'assenza del valore di ritorno: in questo caso, il compilatore segnala l'eventuale errato utilizzo del sottoprogramma all'interno di un'espressione.

Osserva la sintassi:

```
void <NomeProcedura>(<TipoDato1>&<Parametro1>,...,<TipoDatoN>&<ParametroN>)
{
  <Istruzioni>
}
main()
{
  <NomeProcedura>(Parametro1,...,ParametroN)
}
```

La tabella che segue mostra un esempio confrontando le sintassi dei due linguaggi:

C

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

void funzione (int *valRif)
{
  *valRif = *valRif + 5;
}
main()
{
  int x;
  x = 9;
  funzione(&x);
  printf("%d\n",x);
  system("PAUSE");
}
```

C++

```
#include <iostream>
#include <stdlib.h>
using namespace std;
void funzione (int &valRif)
{
  valRif = valRif + 5;
}
main()
{
  int x;
  x = 9;
  funzione(x);
  cout << x;
  system("PAUSE");
}
```

13 I prototipi

Il **prototipo** di una funzione non è altro che una dichiarazione di quella funzione in cui vi è solo la parte dell'intestazione e dove viene indicato solo il tipo degli eventuali parametri formali e non il nome.
Ogni prototipo deve terminare con il simbolo di fine istruzione ;. Ecco la sua sintassi:

```
[<TipoRestituito>] <NomeFunzione>([<tipoParametro>], [<tipoParamentro>],...) ;
```

Ad esempio, il prototipo della funzione void Potenza() vista in un precedente esercizio è:

```
void Potenza(int , int );
```

Nota che il prototipo di una funzione è del tutto simile all'intestazione della sua dichiarazione: la differenza consiste solo nell'omettere il nome dei parametri formali, conservandone invece l'elenco attraverso l'indicazione del tipo.
Solitamente, i prototipi delle funzioni vengono inseriti all'inizio del programma, subito dopo la sezione degli header e comunque prima della chiamata delle funzioni stesse.
Lo scopo dell'utilizzo dei prototipi non è funzionale, ma di puro controllo da parte del compilatore, il quale attraverso di essi verifica la coerenza del numero e del tipo degli argomenti passati con la dichiarazione vera e propria della funzione inserita più avanti nel programma.
Le funzioni che non prevedono parametri formali dovrebbero contenere all'interno del loro prototipo la parola chiave void tra parentesi tonde. Ad esempio, se l'intestazione di una funzione è:

```
void stampa()
```


il suo prototipo è:

```
void stampa(void);
```

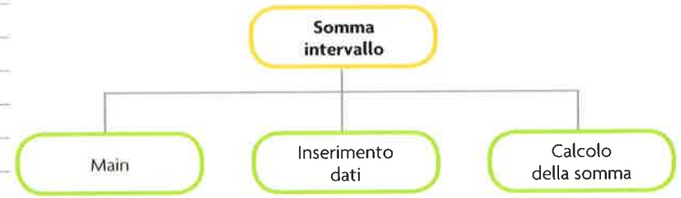
se, nel prototipo, le parentesi tonde sono lasciate vuote, il compilatore non effettua alcun controllo né sul numero né sul tipo degli argomenti eventualmente passati alla funzione.

L'utilizzo dei prototipi consente di spostare la dichiarazione delle funzioni dopo la dichiarazione del main.
Di fatto il compilatore incontra in primis, attraverso i prototipi, un modello di dichiarazione che poi verificherà successivamente con la funzione vera e propria posta in qualsiasi punto del codice.
Proponiamo di seguito un esempio, utilizzando i prototipi e il passaggio sia per valore, sia per riferimento.

OSSERVA COME SI FA

1. **Scrivere un algoritmo che, dati in input due numeri interi positivi non nulli X e Y, visualizzi la somma dei numeri compresi nell'intervallo [X, Y].**

Il problema può essere scomposto in due sottoproblemi: il primo si preoccuperà di richiedere in input i due valori e il secondo di calcolare la somma. Nel calcolo della somma dobbiamo soltanto tenere presente che l'intervallo è chiuso e, di conseguenza, coinvolgere nel calcolo anche gli estremi. Lo schema top-down è pertanto il seguente:



PSEUDOCODICE

```
ALGORITMO SommaIntervallo
Main()
VARIABILI
  X, Y, Somma: INTERO
INIZIO
  InserimentoDati(X, Y)
  Somma ← 0
  Addiziona(X, Y, Somma)
  SCRIVI("La somma dei numeri compresi tra ", X, " e ", Y, " è ", Somma)
FINE
PROCEDURA InserimentoDati(REF A: INTERO; REF B: INTERO)
    // I parametri A e B sono di output
INIZIO
  SCRIVI("Inserisci l'estremo inferiore dell'intervallo")
  LEGGI(A)
  SCRIVI("Inserisci l'estremo superiore dell'intervallo")
  LEGGI(B)
FINE
PROCEDURA Addiziona(VAL K: INTERO; VAL B: INTERO; REF S: INTERO)
     // I parametri A e B sono di input, S è di output
VARIABILI
    I: INTERO
INIZIO
  PER I ← K A B ESEGUI
    S ← S + I
  FINEPER
FINE
```

C++

```
#include <iostream>
using namespace std;
void InserimentoDati( int *, int *);
void Addiziona(int, int, int*);
main()
{
   int X,Y,Somma;
   InserimentoDati(&X,&Y);
   Somma=0;
   Addiziona(X,Y,&Somma);
   cout << "La somma dei numeri compresi tra " << X << " e " << Y << " e\' " << Somma << endl;
   system("PAUSE");
}
void InserimentoDati(int *A, int *B)
{
   cout << "inserisci estremo inferiore intervallo";
   cin >> *A;
   cout << "inserisci estremo superiore intervallo";
   cin >> *B;
}
void Addiziona(int K, int B, int *S)
{
   int i;
   for (i=K; i<B; i++)
      *S+=i;
}
```

14 Le funzioni

Una funzione è un sottoprogramma contenente le istruzioni che risolvono uno specifico problema, e che, quando viene attivato da un'istruzione di chiamata, restituisce un valore. Il valore della funzione deve essere usato come elemento di un'istruzione.

Vediamo subito la differenza con le procedure.

PROCEDURA	FUNZIONE
La procedura permette l'aggregazione di una sequenza di operazioni elementari in una unica macro-operazione dotata di nome e di argomenti. In quanto costrutto sintattico, la procedura introduce regole di visibilità che nascondono a un osservatore esterno la sua struttura interna. Semanticamente essa realizza i suoi effetti modificando parametri ricevuti dal chiamante o un ambiente esterno che può consistere in un insieme di variabili, dispositivi di I/O, file e così via. **La procedura è un'astrazione della nozione di istruzione**: non è altro che un'istruzione complessa che può essere utilizzata ovunque possa esserlo una istruzione semplice: il compito principale di una procedura è quello di modificare il contenuto di locazioni di memoria.	La funzione può essere vista come una procedura con restrizioni di tipo semantico: essa non effettua alcuna azione visibile sul mondo circostante, non modifica gli argomenti ricevuti dal chiamante e realizza i suoi effetti restituendo un risultato. **La funzione è un'astrazione della nozione di operatore** e può essere utilizzata in qualunque valutazione di espressione: ha il compito di fornire un valore, anche se non è escluso che, nel fornirlo, assuma anche i comportamenti della procedura (noi nella nostra trattazione lo escluderemo!).

Ua funzione può essere richiamata in un'assegnazione a una variabile oppure all'interno di una generica espressione. Per questi motivi, e anche per evitare confusione, nell'implementazione dei nostri pseudocodici utilizzeremo la nuova pseudoistruzione:

```
RITORNO (<NomeVariabile>)
```

da inserire, generalmente, prima della pseudoistruzione FINE che chiude il sottoprogramma. Vediamo la sintassi:

```
FUNZIONE <NomeFunzione> ([<ListaParametri>]): <TipoRestituito>
INIZIO
   <Istruzioni>
   RITORNO (<Risultato>)
FINE
```

L'intestazione della funzione differisce un po' da quella della procedura:

```
FUNZIONE <NomeFunzione> ([<ListaParametri>]): <TipoRestituito>
```

dove:

- <NomeFunzione> è il nome associato alla funzione;
- <ListaParametri> costituisce la lista di parametri completi di tipo, necessari per lo scambio delle informazioni con il programma chiamante; come per le procedure, se la funzione non dovesse contenere parametri, il <NomeFunzione> va comunque seguito da una parentesi tonda aperta e da una chiusa ();
- <TipoRestituito> indica quale tipo di valore deve restituire la funzione.

Ora risolviamo un semplice problema che fa uso delle funzioni:

Scrivere un algoritmo che, data in input una sequenza di N numeri interi chiusa dallo 0, calcoli per ciascuno di essi il fattoriale.

Dalla matematica sappiamo che il fattoriale di un numero naturale N maggiore o uguale a zero, denotato con N!, è dato da:

$$0! = 1$$

$$N! = N \times (N - 1) \times (N - 2) \times \cdots \times 2 \times 1$$

Ad esempio, il fattoriale di $5 = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

Passiamo all'algoritmo:

PSEUDOCODICE

```
ALGORITMO CalcoloFattoriale
Main()
VARIABILI
   Num: INTERO
INIZIO
   SCRIVI("Inserisci un numero (0 per finire)")
   LEGGI(Num)
   MENTRE(Num ≠ 0) ESEGUI
      SCRIVI("Il fattoriale di ", Num, " è ", Fattoriale(Num)) // Viene chiamata la funzione Fattoriale
      SCRIVI("Inserisci un numero (0 per finire)")
      LEGGI(Num)
   FINEMENTRE
FINE
```

LO SAI CHE...

Per capire che cos'è una funzione in un programma, basta pensare a una funzione matematica, dove c'è la variabile indipendente x e quella dipendente y. Quest'ultima di solito è il risultato che si trova elaborando la prima variabile, come $y = f(x)$.

La funzione può essere utilizzata nelle espressioni come una semplice variabile, ma non può mai comparire in un'istruzione di lettura o a sinistra in un'istruzione di assegnazione. Ad esempio, se in un programma è stata definita la funzione di nome Prod(), l'istruzione:

```
Tot ← Prod() * N
```

risulta valida e viene interpretata come: "assegna alla variabile Tot il prodotto che si ottiene dal risultato fornito dalla funzione Prod per il valore della variabile N".

```
FUNZIONE Fattoriale(VAL N: INTERO): INTERO
VARIABILI
  Prodotto, I: INTERO
INIZIO
  Prodotto ← 1
  PER I ← 2 A N ESEGUI
    Prodotto ← Prodotto * I
  FINEPER
  RITORNO(Prodotto)  // Il controllo torna all'istruzione successiva alla chiamata
FINE
```

Abbiamo utilizzato una funzione di nome *Fattoriale* che, per ogni numero intero inserito, restituisce il suo fattoriale (anch'esso ovviamente intero).
Ciò giustifica la seguente intestazione utilizzata:

```
FUNZIONE Fattoriale(VAL N: INTERO): INTERO
```

Il risultato viene restituito dalla funzione per mezzo dell'istruzione:

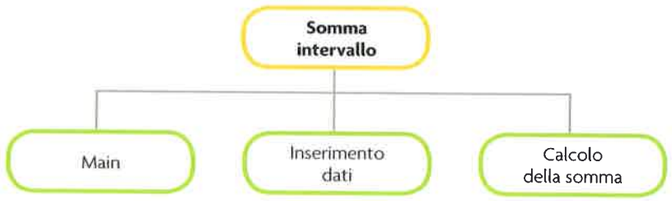
```
RITORNO(Prodotto)
```

dove *Prodotto* è la variabile intera utilizzata per calcolare il fattoriale del singolo numero intero inserito.

OSSERVA COME SI FA

1. Scrivere un algoritmo che, dati in input due numeri interi positivi non nulli X e Y, visualizzi la somma dei numeri compresi nell'intervallo [X, Y].

Questo problema è stato risolto con le procedure. Risolviamolo ancora una volta servendoci, questa volta, di una funzione. Non riportiamo di nuovo l'analisi del problema, poiché è analoga a quella riportata nel precedente esercizio.



PSEUDOCODICE

```
ALGORITMO SommaIntervallo
Main()
VARIABILI
  X, Y: INTERO
INIZIO
  InserimentoDati(X, Y)
  SCRIVI("La somma dei numeri compresi tra ", X, " e ", Y, " è ", Addiziona(X, Y))
FINE
```

```
PROCEDURA InserimentoDati(REF A: INTERO; REF B: INTERO)
INIZIO
  SCRIVI("Inserisci l'estremo inferiore dell'intervallo")
  LEGGI(A)
  SCRIVI("Inserisci l'estremo superiore dell'intervallo")
  LEGGI(B)
FINE

FUNZIONE Addiziona(VAL K: INTERO; VAL Z: INTERO): INTERO
VARIABILI
  I, Somma: INTERO
INIZIO
  Somma ← 0
  PER I ← K A Z ESEGUI
    Somma ← Somma + I
  FINEPER
  RITORNO(Somma)  // Il controllo torna all'istruzione successiva alla chiamata
FINE
```

2. Visualizzare il prodotto di due numeri interi servendosi della sola operazione di addizione.

Il problema è molto semplice: occorre fornire in input due numeri A e B e sommare B volte il numero A. Supponiamo, ad esempio, di voler svolgere il prodotto 5 × 3. Per realizzarlo occorre sommare tre volte il numero cinque. Il problema può essere scomposto in due sottoproblemi, precisamente quello per l'acquisizione dei dati e quello per il calcolo del prodotto. Il top-down è il seguente:

PSEUDOCODICE

```
ALGORITMO Moltiplicazione
Main()
VARIABILI
  A, B: INTERO
INIZIO
  SCRIVI("Inserire il primo numero")
  LEGGI(A)
  SCRIVI("Inserire il secondo numero")
  LEGGI(B)
  SCRIVI("Il prodotto di ", A, " e ", B, " è ", Moltiplica(A, B)) // Si attiva la funzione Moltiplica()
FINE

FUNZIONE Moltiplica(VAL A: INTERO; VAL B: INTERO): INTERO
VARIABILI
  Prod, I: INTERO
```

```
INIZIO
  Prod ← 0          // Variabile locale che conterrà il prodotto di A per B
  PER I ← 1 A B ESEGUI
    Prod ← Prod + A
  FINEPER
  RITORNO(Prod)     // Il controllo torna all'istruzione successiva alla chiamata
FINE
```

ORA TOCCA A TE!

1 Qual è l'output fornito dal seguente algoritmo?

PSEUDOCODICE

```
ALGORITMO Sorpresa
Main()
VARIABILI
  Numero: INTERO
INIZIO
  SCRIVI("Inserisci un numero")
  LEGGI(Numero)
  SCRIVI("Ho restituito ", Esegui(Numero))
FINE
FUNZIONE Esegui(VAL Num: INTERO): INTERO
INIZIO
  RITORNO(Num * 2)
FINE
```

2 Perché il seguente algoritmo per trovare il massimo e il minimo tra due numeri non produce il risultato voluto? Correggi gli errori presenti

PSEUDOCODICE

```
ALGORITMO Errori
Main()
VARIABILI
  Numero!, Numero2: INTERO
INIZIO
  SCRIVI("Inserisci un numero")
  LEGGI(Numero1)
  SCRIVI("Inserisci un altro numero")
  LEGGI(Numero2)
  Minimo(Numero1, Numero2)
  Massimo(Numero1, Numero2)
FINE
FUNZIONE Minimo(VAL N1:INTERO; VAL N2:INTERO)
INIZIO
  SE(N1 ≤ N2)
    ALLORA
      RITORNO(N1)
    ALTRIMENTI
      RITORNO(N2)
  FINESE
FINE
FUNZIONE Minimo(VAL N1:INTERO; VAL N2:INTERO)
INIZIO
  SE(N1 ≤ N2)
    ALLORA
      RITORNO(N1)
    ALTRIMENTI
      RITORNO(N2)
  FINESE
FINE
```

3 Descrivi la logica e il comportamento della seguente funzione:

PSEUDOCODICE

```
FUNZIONE Bisestile(VAL Anno: INTERO): BOOLEANO
INIZIO
  SE(Anno MOD 4 = 0)
   ALLORA
      SE(Anno MOD 100 = 0)
      ALLORA
         SE(Anno MOD 1000 = 0)
         ALLORA
            Bisestile ← VERO
         ALTRIMENTI
            Bisestile ← FALSO
      ALTRIMENTI
         Bisestile ← VERO
   ALTRIMENTI
      Bisestile ← FALSO
FINE
```

15 Le funzioni in C/C++

Una funzione, come si è detto, differisce da una procedura sostanzialmente perché ritorna un valore al programma chiamante.
Per dichiarare una funzione si utilizza la sintassi seguente:

```
FUNZIONE <NomeFunzione>([<ListaParametri>]):
INIZIO
  <Istruzioni>
  RITORNO (<Risultato>)
FINE
```

```
<TipoRestituito><NomeFunzione>([<ListaParametri>])
{
  <Istruzioni>
  return [(]<Risultato>[)]
}
```

Il main è una funzione come tutte le altre, che può restituire un valore al sistema operativo; è quindi buona norma che il programma restituisca il valore 0 se viene eseguito fino alla fine senza errori. Da qui in avanti, tutti i nostri programmi C/C++ riporteranno l'intestazione della funzione main come int main() e la sua ultima istruzione sarà return 0;

Fra le istruzioni contenute nella dichiarazione di funzione assume una particolare importanza **return**, utilizzata per restituire un valore al chiamante. La sua sintassi prevede di specificare, dopo la parola chiave **return**, un valore costante o una variabile o un'espressione, compatibili con il tipo restituito dalla funzione.
Ad esempio:

```
return 2;        ← Restituisce il valore 2; si può anche scrivere: (2)
return x;        ← Restituisce il valore 2 contenuto nella variabile x
return (a+b);    ← Restituisce il risultato dell'espressione a + b
```

Supponiamo di voler calcolare la media dei pesi di un gruppo di amici e di demandare il conteggio totale dei pesi a una funzione:

C

```
#include <stdio.h>
#include <stdlib.h>
#define PESI 5
float calcola()                    ← Intestazione della funzione calcola
{
  int i;                           ← Definizione della funzione calcola
  float peso,pesoTotale;
  for(i=0;i<PESI;i++)
  {
    printf("inserisci il peso");
    scanf("%f",&peso);
    pesoTotale+=peso;
  }
  return pesoTotale;               ← Valore di ritorno dalla funzione calcola
}
int main()
{
  float media;
  media=calcola()/PESI;            ← Chiamata della funzione calcola all'interno di un'espressione
  printf("La media corrisponde a %f",media);
  system("PAUSE");
  return 0;
}
```

C++

```
#include <iostream>
using namespace std;
  #define PESI 5
  float calcola()
  {
    int i;
    float peso,pesoTotale;
    for(i=0;i<PESI;i++)
    {
      cout << "inserisci il peso";
      cin >> peso;
      pesoTotale+=peso;
    }
  return pesoTotale;
  }
  int main()
  {
    float media;
    media=calcola()/PESI;
    cout << "La media corrisponde a " << media;
    system("PAUSE");
    return 0;
  }
```

Nell'esempio precedente, la funzione calcola() è chiamata all'interno di un'espressione; poiché, come si è detto, una funzione ritorna un valore, questo deve essere assegnato a una variabile o a una costante, o comunque utilizzato all'interno dell'espressione.
In particolare, **il tipo del valore di ritorno deve coincidere con il tipo attribuito all'identificatore della funzione**. Nel nostro caso ciò avviene correttamente, perché sono entrambi di tipo float.
Osserva che sarebbe stato possibile inserire la chiamata alla funzione anche direttamente all'interno dell'istruzione printf/cout:

```
printf("La media corrisponde a %f", calcola()/PESI);
```

```
cout << "La media corrisponde a " << media;
```

in questo modo avremmo evitato di dichiarare e utilizzare la variabile media.

16 La ricorsività

Consideriamo una funzione matematica ben nota: la **potenza** di un numero relativo o reale diverso da 0 (con un numero naturale come esponente):

$a^n = a \cdot a \cdot \ldots \cdot a$ (n volte) se $n > 0$; $a^n = 1$ se $n = 0$

Possiamo definirla anche nel modo seguente:

$$\begin{cases} a^n = 1 \text{ se } n = 0 \text{ e } a \neq 0 \\ a^n = a \cdot a^{n-1} \text{ se } n > 0 \end{cases}$$

in base al quale:

$a^{n-1} = a \cdot a^{n-2}$
$a^{n-2} = a \cdot a^{n-1}$
...

Una caratteristica apparentemente paradossale di questa definizione è il fatto che essa definisce la potenza ricorrendoalla stessa definizione di potenza! Infatti, la potenza di un numero compare sia a sinistra sia a destra del segno di uguaglianza. Poniamo dunque la seguente definizione:

si dice ricorsiva (diretta) una funzione al cui interno compare una chiamata a se stessa.

Dal punto di vista della programmazione, una funzione può, in effetti, chiamare se stessa, perché a ogni chiamata viene generata nella memoria centrale una copia della funzione, come se si trattasse di una funzione diversa: il codice relativo viene caricato in memoria al momento della

LO SAI CHE...

Contrariamente a quanto si potrebbe pensare, tuttavia, una funzione ricorsiva non riduce le dimensioni del programma eseguibile e nemmeno della memoria che il programma stesso occupa; anzi, generalmente si ha una riduzione della velocità di esecuzione, a causa delle continue chiamate che impegnano il sistema operativo a creare ed eliminare in continuazione le variabili nello **stack** (l'area di memoria nella quale viene allocato lo spazio per le variabili locali, compresi i parametri formali di un sottoprogramma).

chiamata e rilasciato appena termina l'esecuzione della funzione; questa attività del sistema operativo, nella gestione della memoria centrale, è detta **allocazione dinamica del codice**.
Il linguaggio C/C++ ben si presta all'implementazione della ricorsività. Vediamo ad esempio un programma che implementa la definizione ricorsiva di potenza vista qui.

OSSERVA COME SI FA

1. Calcolare la potenza di un numero con il metodo ricorsivo.

C

```c
#include <stdio.h>
#include <stdlib.h>
int potenza(int,int);
int a,n;
int main()
{  printf("inserisci base ");
   scanf("%d",&a);
   printf("inserisci esponente ");
   scanf("%d",&n);
   printf(" %d elevato a %d =%d \n",a,n,potenza(a,n));
   system("PAUSE");
   return 0;
}
int potenza(int x, int y)
   { if( y==0) return 1;
     else return x*potenza(x,y-1);
}
```

Chiamata ricorsiva

C++

```cpp
#include <iostream>
using namespace std;
int potenza(int,int); //prototipo
int a,n;
int main()
{
   cout << "inserisci base ";
   cin >> a;
   cout << "inserisci esponente ";
   cin >> n;
   cout << a << " elevato a " << n << "=" << potenza(a,n) << endl;
   system("PAUSE");
   return 0;
}
int potenza(int x, int y)
{ if( y==0) return 1;
     else return x*potenza(x,y-1);
}
```

```
inserisci base 8
inserisci esponente 2
8 elevato a 2=64
Premere un tasto per continuare . . .
```


2. Calcolare il fattoriale di un numero intero fornito in input.

In ambito matematico, esistono numerose funzioni ricorsive di grande importanza. Un altro esempio è dato dal **fattoriale** di un numero naturale n, definito ricorsivamente con:

$$\begin{cases} n! = 1 & \text{se } n = 0; \\ n! = (n-1)! \cdot n & \text{se } n > 0 \end{cases}$$

Per esempio: $7! = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5040$. Un programma che implementa questa definizione è il seguente:

C

```c
#include <stdio.h>
#include <stdlib.h>
int fattoriale(int);
int a;
int main()
{  printf("inserisci il numero per il calcolo del fattoriale");
   scanf("%d",&a);
   printf("fattoriale di %d = %d \n",a,fattoriale(a));
   system("PAUSE");
   return 0;
}
int fattoriale(int y)
{  if( y==0) return 1;
   else return y*fattoriale(y-1);
}
```

```
inserisci il numero per il calcolo del fattoriale 8
fattoriale di 8 = 40320
Premere un tasto per continuare . . .
```

C++

```cpp
#include <iostream>
using namespace std;
int fattoriale(int);
int main()
{
   int a;
   cout << "inserisci il numero per il calcolo del fattoriale ";
   cin >> a;
   cout << "fattoriale di " << a << " = " << fattoriale(a) << endl;
   system("PAUSE");
   return 0;
}
int fattoriale(int y)
{ if( y==0) return 1;
     else return y*fattoriale(y-1);
}
```

Sulla base di questi esempi possiamo riconoscere che, per attivare un procedimento ricorsivo:

- il problema deve essere scomponibile in sottoproblemi dello stesso tipo, ognuno dipendente dall'altro in base a una scala gerarchica;
- si devono conoscere le **relazioni funzionali** che legano il problema ai sottoproblemi simili (**caso generale**);
- è necessario conoscere la soluzione di (almeno) un **caso particolare** del problema, indispensabile per terminare il problema stesso (**condizione di terminazione** o **casi base** o **clausole di chiusura**).

Possiamo quindi individuare tre elementi caratteristici dei "problemi ricorsivi":

1. una **condizione**, che permette di verificare se si è di fronte a un caso particolare risolvibile banalmente o se è necessario ripetere il procedimento di soluzione;
2. il procedimento di soluzione del **caso generale**, che contiene una o più chiamate ricorsive;
3. la soluzione di (almeno un) **caso particolare** (che termina la soluzione).

TRAINING

Test

CONOSCENZE

Top-down

1. Stabilisci se le seguenti affermazioni sono vere o false.

a La tecnica top-down nasce come tecnica di progettazione dei problemi. | V | F |

b Nella tecnica top-down il problema viene scomposto. | V | F |

c La tecnica top-down non è indicata per la programmazione imperativa. | V | F |

2. A differenza di una procedura, una funzione:

a è più veloce
b risulta meno leggibile
c non ha variabili locali
d restituisce un valore

Visibilità e durata delle variabili

3. L'ambiente locale di un sottoprogramma è costituito:

a dalle risorse dichiarate all'interno del sottoprogramma
b dalle risorse globali
c dalle risorse locali
d da tutti i sottoprogrammi presenti nell'algoritmo

4. L'ambiente globale di un sottoprogramma è costituito:

a dalle risorse dichiarate dentro al sottoprogramma
b dalle risorse globali
c dalle risorse locali
d da tutti i sottoprogrammi presenti nell'algoritmo

5. Una variabile locale:

a è utilizzabile solo all'interno del sottoprogramma in cui è stata dichiarata
b non viene deallocata all'uscita dal sottoprogramma
c viene deallocata all'uscita dal programma
d è utilizzabile anche da sottoprogrammi dichiarati internamente al sottoprogramma

Parametri e prototipi

6. Stabilisci se le seguenti affermazioni sono vere o false.

a I parametri attuali sono quelli ricevuti dal sottoprogramma. | V | F |

b I parametri formali sono quelli inviati al sottoprogramma. | V | F |

c Ci deve essere sempre costante accordo tra numero, tipo e ordine dei parametri attuali e numero, tipo e ordine di quelli formali. | V | F |

7. L'assegnazione c=calcola(x); con *c* di tipo int è corretta se:

a **calcola** è una procedura
b la funzione **calcola** restituisce un **int**
c la variabile *x* è di tipo **int**
d il tipo della variabile *c* è più piccolo di quello della funzione

8. Un prototipo di funzione:

a è necessario al funzionamento del programma
b stabilisce la coerenza con la rispettiva funzione
c fornisce i parametri formali alla funzione
d modifica il valore ritornato dalla funzione

La ricorsività

9. Quale tra i seguenti è un vantaggio degli algoritmi ricorsivi?

a sono più facili da realizzare
b sono più difficili da capire da parte di altri programmatori che non siano autori di quel software
c sono più facili da testare
d sono esteticamente più belli

COMPETENZE

Procedure senza parametri

1. Scrivi un algoritmo che, servendosi di un menu, richiami tre procedure attraverso le quali visualizzare, rispettivamente, i dati della scuola, i dati della propria abitazione, i dati della palestra che si frequenta.

2. Scrivi un algoritmo che, servendosi di un menu, richiami tre procedure attraverso le quali visualizzare, rispettivamente, i numeri da 1 a 20, i primi 20 numeri pari e i primi 20 numeri dispari.

3. Scrivi un algoritmo contenente una procedura che stampa la tavola pitagorica.

Procedure con parametri

4. Scrivi un algoritmo composto da una procedura che riceve un numero intero e visualizzi in output tanti asterischi quanti sono indicati dal parametro trasmesso. Il numero intero non può essere maggiore di 10.

5. Scrivi un algoritmo composto da una procedura che riceve un numero intero minore o uguale a dodici e visualizzi il corrispondente in lettere.

6. Scrivi un algoritmo composto da una procedura che riceve un numero reale e visualizzi separatamente la parte intera e quella decimale.

Procedure con passaggio di parametri

7. Scrivi un algoritmo che richiesti in input il nome e l'età di tre persone visualizzi i dati in ordine alfabetico e in ordine di età.

8. Scrivi un algoritmo che riceve in input il nome e l'età di una serie di N persone (N ≤ 10). Visualizzare su richiesta dell'utente la media dell'età, la persona più giovane, la persona più grande.

9. Scrivi un algoritmo che, a scelta dell'utente, realizzi la moltiplicazione o la divisione tra due numeri interi positivi servendosi solo dell'operazione di addizione.

10. Scrivi un algoritmo che, dati in input due numeri interi positivi, determini il MCD e il mcm.

11. Scrivi un algoritmo che, data in input una sequenza di numeri chiusa dallo zero, determini tutti i numeri che risultano maggiori della somma di tutti i precedenti.

12. Scrivi un algoritmo per la gestione di un conto corrente. Su tale conto possono essere svolte tre tipi di operazioni: versamento, prelevamento, emissione di assegni. Dopo aver introdotto il saldo iniziale, si potranno inserire le varie operazioni digitandone il tipo e la somma. A richiesta dell'utente, l'algoritmo dovrà fornire:

- il numero dei versamenti effettuati e la somma totalmente versata;
- il numero dei prelevamenti effettuati e la somma totalmente prelevata;
- il numero degli assegni emessi e la somma totalmente prelevata per mezzo degli stessi;
- il saldo finale.

13. Realizza una funzione che calcoli la potenza *n*-esima di una base data.

14. Scrivi un programma che, utilizzando le funzioni, implementi il gioco nel quale l'utente deve indovinare un numero segreto entro un numero massimo di tentativi.

15. Scrivi un programma che, sfruttando le funzioni, permetta di calcolare l'area di un cerchio o di un quadrato. L'utente inserisce un numero, *dichiarando* se si tratta del raggio di un cerchio o del lato di un quadrato. Se l'utente inserisce un numero negativo viene visualizzato un errore, altrimenti il sistema calcola l'area in modo appropriato.

16. Scrivi una procedura che calcoli la parte intera della radice quadrata di un numero intero ricevuto in input.

17. Scrivi una procedura che risolva un sistema lineare di due equazioni in due incognite:

$a_1x + b_1y = c_1$
$a_2x + b_2y = c_2$

utilizzando le equazioni:

$x = (c_1b_2 - c_2b_1) / (a_1b_2 - a_2b_1)$
$y = (a_1c_2 - a_2c_1) / (a_1b_2 - a_2b_1)$

(*Suggerimento:* poni $c_1b_2 - c_2b_1 = XN$, $a_1c_2 - a_2c_1 = YN$, $a_1b_2 - a_2b_1 = D$ e per le incognite dichiara delle variabili globali.)

Ricorsività

18. Realizza un programma che calcoli il quoziente della divisione tra interi. Svolgi l'esercizio nelle due versioni, iterativa e ricorsiva.

19. Scrivi una funzione C/C++ che calcoli, dati due numeri interi M e N, la potenza M^N. Svolgi l'esercizio nelle due versioni, iterativa e ricorsiva.

20. Variante dell'esercizio precedente: estendi la funzione al caso di M reale e cerca di minimizzare il numero dei prodotti che vengono effettuati, sfruttando opportunamente la seguente osservazione:
se N è dispari, allora $M^N = (M^N - 1) \times M$;
se N è pari, allora $M^N = (M^{N/2})^2$.

21. Scrivi un programma C/C++ che, data in input una sequenza di numeri chiusa da zero, determini tutti i numeri maggiori della somma di tutti i precedenti.

22. Scrivi un programma C/C++ per la gestione di un conto corrente. Su tale conto possono essere effettuate tre tipi di operazioni: versamento, prelievo, emissione assegni. Dopo aver introdotto il saldo iniziale, si potranno inserire le varie operazioni digitandone il tipo e la somma interessata. A richiesta dell'utente, il programma dovrà fornire:

- il numero dei versamenti effettuati e la somma totalmente versata;
- il numero dei prelevamenti effettuati e la somma totalmente prelevata;
- il numero degli assegni emessi e la somma totalmente prelevata per mezzo degli stessi;
- il saldo finale.