Brevissima introduzione al C++

Prof. Marco Marfurt

Dipartimento di Matematica

UNIVERSITÀ DEGLI STUDI DI ROMA
LA SAPIENZA

Indice

Istruzioni per l'uso

1. **Che cosa sono queste dispense:**

   Queste dispense sono un aiuto pensato per chi ha nessuna o pochissima esperienza nell'uso di un linguaggio di programmazione; esse sono essenzialmente costituite da una sequenza di programmi in C++, ciascuno dei quali è ampiamente commentato e ciascuno dei quali introduce istruzioni e tecniche di programmazione di livello di complicazione crescente. Il modo corretto per usarle è quello di sedersi davanti al computer e di leggere il testo dalla prima all'ultima pagina e, ogni volta che si incontra un esempio di programma, provare a scriverlo sul computer, a compilarlo e ad eseguirlo, verificando che cosa succede. Una volta compreso bene come funziona un certo programma, un utile esercizio può essere quello di provare a modificarlo in modo da fargli fare cose diverse; ciò favorirà lo sviluppo di una capacità autonoma di produrre programmi. Dopo avere studiato queste dispense, dovreste essere in grado di realizzare dei programmi in C++ di moderata complessità in maniera autonoma. Non vi preoccupate troppo se rimarrà ancora qualche punto oscuro o qualche confusione o incertezza; avremo tempo in seguito per chiarire meglio le cose.

2. **Che cosa non sono queste dispense:**

   Queste dispense non sono un manuale di C++; non pensate quindi di trovarvi tutte le possibili istruzioni del linguaggio C++ e neanche tutte le sue regole grammaticali e sintattiche riportate in maniera chiara e completa. Per questo dovete leggere un vero manuale di C++ (ne esistono tantissimi in commercio o consultabili nella nostra biblioteca del Dipartimento di Matematica). Il mio consiglio è quello di studiare e capire prima di tutto queste dispense; successivamente potrete procurarvi un manuale di C++ e studiarlo.

3. **Quale compilatore usare:**

   I programmi riportati in queste dispense sono stati scritti pensando di usare il compilatore Dev-C++ che è installato nel Laboratorio di Calcolo del nostro Dipartimento, ma che può essere prelevato liberamente in rete ed installato facilmente su qualsiasi PC anche a casa vostra (cosa che caldamente vi raccomando di fare). Nel caso vogliate usare un altro compilatore potrebbe essere necessario apportare ai programmi qualche piccola modifica.

Capitolo 1

I primi programmi in C++: ingresso e uscita di dati

Ecco un primo esempio di programma in linguaggio C++:

```
 Esempio 1.1
  /* QUESTO PROGRAMMA MOLTIPLICA 3 PER 5 E SOMMA 7 AL
   RISULTATO DEL PRODOTTO */
#include <stdio.h>
#include <stdlib.h>
 main()
{
   int x,y;
   x=5;
   y=3*x+7;
   printf("%d",y);
   system ("PAUSE");
   return 0;
}
```

Analizziamolo riga per riga per cominciare a capire le regole con cui è stato scritto: prima di tutto, le prime due righe

```
/* QUESTO PROGRAMMA MOLTIPLICA 3 PER 5 E SOMMA 7 AL
   RISULTATO DEL PRODOTTO */
```

sono un commento; in generale tutto ciò che compare scritto fra il segno /* e il segno */ è un commento, esso viene ignorato dal compilatore e quindi non fa propriamente parte del programma C++. I commenti servono solo a chi legge il programma per suggerirgli che cosa questo faccia; i commenti

non sono obbligatori, tuttavia sono caldamente raccomandati per favorire la lettura del programma da parte di chiunque. In questo esempio e nei successivi si è scelto di scrivere i commenti in caratteri maiuscoli; anche questo non è obbligatorio, ma è solo una scelta per renderli più visibili. Passiamo ora alla terza riga:

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

anche questa riga non fa parte propriamente del programma C++; essa è una direttiva che avverte il compilatore che una o più istruzioni che compariranno in seguito, si possono trovare nel file **stdio.h**, e i simboli < e > servono ad indicare che questo particolare file si trova nella Libreria Standard.

Analogo discorso vale per la quarta riga:

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

Veniamo ora alla quinta riga

```
main()
```

che avverte che da qui inizia il programma C++ vero e proprio e che tale programma è costituito dalle istruzioni comprese fra il segno '**{**' e il segno '**}**'. Il segno '**;**' serve ad indicare dove finisce ciascuna istruzione e quindi non sarebbe necessario andare a capo dopo ogni '**;**' come nell'esempio sopra riportato, e il programma potrebbe essere scritto equivalentemente nella forma

```
main(){int x,y;x=5;y=3*x+7;printf("%d",y);
system ("PAUSE");return 0;}
```

anche se uno stile di programmazione di questo tipo non è raccomandabile per ragioni di leggibilità.

Vediamo ora di analizzare una per una le singole istruzioni:

Istruzione 1: int x,y; Questa istruzione serve a dichiarare che x e y sono due variabili di tipo intero; dal punto di vista pratico, il compilatore individuerà due aree di memoria di dimensione opportuna (in questo caso di 4 bytes ciascuna, ma con altri compilatori potrebbe essere di 2 bytes) che da questo momento verranno riservate alle variabili x ed y rispettivamente.

Istruzione 2: x=5; Questa istruzione assegna alla variabile x il valore 5.

Istruzione 3: y=3*x+7; Questa istruzione moltiplica per 3 il valore attuale della variabile x (che è 5), gli somma 7 ed assegna il numero così ottenuto (cioè 22) come valore della variabile y.

Istruzione 4: printf("%d",y); Questa istruzione serve a stampare (sullo schermo) il valore della variabile y con il formato specificato, cioè il formato **%d** che è il formato per i numeri interi. Per eseguire la funzione **printf**, il compilatore avrà bisogno di fare riferimento al file **stdio.h** (che è appunto il file in cui è definita questa funzione), e questa è la ragione per cui abbiamo dovuto includerlo mediante la direttiva

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

che è quindi assolutamente necessaria (se non ci fosse, il compilatore dichiarerebbe un errore).

Istruzione 5: system ("PAUSE"); Questa istruzione serve a fissare lo schermo nella situazione attuale in modo che noi possiamo leggervi il risultato.

Istruzione 6: return 0; Questa istruzione serve a terminare il programma e a restituire il controllo al sistema operativo del computer.

Osserviamo che il segno '=' funziona come operatore di assegnazione, e precisamente assegna alla variabile che si trova a sinistra del segno '=' il valore che si ottiene sulla destra. Come conseguenza di ciò, il programma precedente potrebbe equivalentemente essere scritto nella forma seguente:

Esempio 1.2

```
/* QUESTO PROGRAMMA MOLTIPLICA 3 PER 5 E SOMMA 7 AL
  RISULTATO DEL PRODOTTO */
#include <stdio.h>
#include <stdlib.h>
 main()
{
  int x;
  x=5;
  x=3*x+7;
  printf("%d",x);
  system ("PAUSE");
  return 0;
}
```

Il risultato pratico sullo schermo del programma dell'Esempio 1.1 e dell'Esempio 1.2 saranno assolutamente identici (provare per credere!). Immaginiamo ora di eseguire il programma dell'Esempio 1.1 (o dell'Esempio 1.2); vedremo comparire sulla prima riga dello schermo

```
 22Premere un tasto per continuare...
```

e se a questo punto premiamo un tasto qualsiasi, lo schermo di uscita sparisce.

Osserviamo ora il seguente programma:

```
Esempio 1.3
// QUESTO PROGRAMMA MOLTIPLICA 3 PER 5 E SOMMA 7
// AL RISULTATO DEL PRODOTTO
 #include <stdio.h>
 #include <stdlib.h>
 main()
 {
   int x,y;
   x=5;
   y=3*x+7;
   printf("\n");
   printf("%d",y);
   system ("PAUSE");
   return 0;
 }
```

Questo programma è quasi identico a quello dell'Esempio 1.1, con due piccole differenze: la prima è che il commento è scritto usando la doppia barra '//' anziché '/*' e '*/'; anche questo stile di commento è comunemente usato, si osservi però che la doppia barra '//' dichiara come commento solo ciò che si trova sulla stessa riga dopo la '//' e il suo effetto termina andando a capo; per questo anche la seconda riga del commento deve avere la sua '//'. D'altra parte con la '//' non c'è bisogno di indicare dove finisce il commento, perché esso termina automaticamente andando a capo. L'altra differenza rispetto al programma dell'Esempio 1.1, è che è stata aggiunta l'istruzione

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

il cui effetto è semplicemente quello di far andare a capo (sullo schermo). Se eseguiamo il programma dell'Esempio 1.3, vedremo comparire sullo schermo la prima riga vuota e sulla seconda riga di nuovo:

```
 22Premere un tasto per continuare...
```

L'istruzione per andare a capo può essere ripetuta più volte semplicemente aggiungendo altri

```
\n
```

all'interno delle virgolette nella istruzione **printf**; per esempio, nel programma seguente

```
Esempio 1.4
 /* QUESTO PROGRAMMA MOLTIPLICA 3 PER 5
E SOMMA 7 AL RISULTATO DEL PRODOTTO */
  #include <stdio.h>
  #include <stdlib.h>
   main()
  {
   int x,y;x=5;y=3*x+7;
   printf("\n\n\n\n");
   printf("%d",y);
   printf("\n\n\n\n");
   system ("PAUSE");
   return 0;
}
```

viene chiesto di andare a capo 4 volte prima di stampare il valore di y e 4 volte dopo.

Per comprendere meglio le potenzialità dell'istruzione **printf**, può essere utile eseguire il seguente programma

```
 Esempio 1.5
// QUESTO PROGRAMMA MOLTIPLICA 3 PER 5 E SOMMA 7
// AL RISULTATO DEL PRODOTTO
 #include <stdio.h>
 #include <stdlib.h>
  main()
 {
   int x,y;x=5;y=3*x+7;
   printf("\n\n\n\n   3x5+7=%d\n\n\n\n",y);
   system ("PAUSE");
   return 0;
 }
```

Vediamo ora una nuova istruzione che è, in qualche modo, l'inversa di **printf**; questa nuova istruzione si chiama **scanf** ed è l'inversa di **printf** nel senso che, come **printf** serviva a trasferire una informazione posseduta dal calcolatore (cioè il valore della variabile y) a me attraverso lo schermo, così **scanf** serve per trasferire una informazione posseduta da me (per esempio il valore che io voglio inizialmente assegnare ad una variabile) al calcolatore, e il trasferimento di informazione avviene, in questo caso, attraverso la tastiera. Ecco un primo semplice esempio di come possiamo utilizzare la nuova istruzione **scanf**:

```
Esempio 1.6
 /* QUESTO PROGRAMMA MOLTIPLICA 3 PER
 UN NUMERO QUALSIASI FORNITO IN LETTURA ATTRAVERSO LA
 TASTIERA E SOMMA 7 AL RISULTATO DEL PRODOTTO */
#include <stdio.h>
#include <stdlib.h>
 main()
{
  int x,y;
  printf("\n\n\n\n   x=");scanf("%d",&x);
  y=3*x+7;
  printf("   3x%d+7=%d\n\n\n\n",x,y);
  system ("PAUSE");
  return 0;
}
```

In questo esempio, l'istruzione

```
printf("\\n\\n\\n\\n x=");
```

ha l'effetto di far comparire sullo schermo 4 righe vuote e poi

```
x=
```

e a questo punto l'istruzione **scanf** ferma lo schermo in questa posizione, in attesa che (attraverso la tastiera) io fornisca al calcolatore il valore della variabile x che compare in **scanf**; se io digito sulla tastiera il numero 5 e poi premo il tasto Invio (detto anche tasto Return), viene assegnato il valore 5 alla variabile x e vengono eseguite le successive istruzioni, per cui, alla fine, comparirà sullo schermo il risultato finale

```
3x5+7=22.
```

C'è una osservazione importante da fare subito a proposito dell'Esempio 1.6 e dell'uso di **scanf**, ed è la seguente: nei programmi precedenti, veniva calcolato il risultato di *3 moltiplicato per 5 e sommato a 7* (che forniva invariabilmente 22), mentre ora viene calcolato il risultato di *3 moltiplicato per il valore della variabile x e sommato a 7*, che fornisce un risultato che dipende dal valore assegnato ad x tramite la tastiera; se io digito sulla tastiera il numero 6 (anziché il 5) e poi premo il tasto Invio, viene assegnato il valore 6 alla variabile x e vengono eseguite le successive istruzioni, per cui, alla fine, comparirà sullo schermo il risultato finale

```
3x6+7=25.
```

In altre parole, il programma dell'Esempio 1.6 permette di calcolare, per un valore di x qualsiasi, la funzione lineare

$$y = 3x + 7.$$

Ecco come possiamo estendere l'uso di **scanf** per assegnare tramite la tastiera il valore di tutti e tre i numeri che compaiono nei programmi precedenti:

```
 Esempio 1.7
  /* QUESTO PROGRAMMA MOLTIPLICA DUE NUMERI QUALSIASI
 a ED x FORNITI IN LETTURA ATTRAVERSO LA TASTIERA E
 SOMMA AL RISULTATO DEL PRODOTTO UN TERZO NUMERO b
 FORNITO IN LETTURA ATTRAVERSO LA TASTIERA */
#include <stdio.h>
#include <stdlib.h>
 main()
 { int a,b,x,y;
   printf("\n\n\n\n   a=");scanf("%d",&a);
   printf("\n   x=");scanf("%d",&x);
   printf("\n   b=");scanf("%d",&b);
   y=a*x+b;
   printf("\n   %dx%d+%d=%d\n\n\n\n",a,x,b,y);
   system ("PAUSE");return 0; }
```

Ecco infine un programma in cui utilizziamo l'istruzione **printf** per presentare elegantemente il risultato del calcolo:

Esempio 1.8

```
 /* QUESTO PROGRAMMA MOLTIPLICA 3 PER UN NUMERO
    QUALSIASI FORNITO IN LETTURA ATTRAVERSO LA
    TASTIERA E SOMMA 7 AL RISULTATO DEL PRODOTTO */
 #include <stdio.h>
 #include <stdlib.h>
  main()
 {int x,y;
 printf(" QUESTO PROGRAMMA MOLTIPLICA 3 PER UN NUMERO QUALSIASI");
 printf("\n FORNITO IN LETTURA ATTRAVERSO LA TASTIERA E SOMMA 7 AL");
 printf("\n RISULTATO DEL PRODOTTO.");
 printf("\n\n\n\n        dammi un numero x a tua scelta");
       printf(" e poi premi il tasto <return>\n\n");
 printf("                              x=");
 scanf("%d",&x);
 y=3*x+7;printf("\n");
       printf("                          ******************");
       printf("\n                           *ECCO IL RISULTATO*        ");
 printf("\n                          ******************");
       printf("\n\n\n");
 printf("                 ******************************");
 printf("\n                 *                             *");
 printf("\n                 *                             *");
 printf("\n                 *         3x%d+7=%d            *",x,y);
 printf("\n                 *                             *");
 printf("\n                 *                             *");
 printf("\n                 ******************************\n\n\n");
 system ("PAUSE");return 0; }
```

Analizziamo ora il seguente programma:

Esempio 1.9

```
/* QUESTO PROGRAMMA SCAMBIA I VALORI DI DUE VARIABILI
   IN MODO SBAGLIATO */
#include <stdio.h>
#include <stdlib.h>
 main()
 {
   int a,b;
   printf("\n  valori di a e b prima dello scambio");
   printf( "\n\n    a=") ; scanf("%d",&a);
   printf( "\n    b=") ; scanf("%d",&b);
   a=b;b=a; // ECCO DOVE STA LO SBAGLIO
   printf("\n\n\n  valori di a e b dopo lo scambio");
   printf("\n\n     a=%d   b=%d",a,b);
   system ("PAUSE");
   return 0;
 }
```

Il programma dell'Esempio 1.9 dovrebbe leggere da tastiera i valori di due variabili intere (a e b) e dovrebbe poi scambiarne i valori, cioè il valore che inizialmente avevamo assegnato alla variabile a dovrebbe passare alla variabile b e il valore assegnato inizialmente alla variabile b dovrebbe passare alla variabile a; tuttavia lo scambio di valori avviene (come ci avverte il commento) in modo sbagliato: vediamo di capirne il perché. Supponiamo di avere assegnato (tramite la tastiera) ad a il valore 1 e a b il valore 2; lo scambio *dovrebbe* avvenire mediante le due istruzioni

```
a=b;b=a;
```

tuttavia la prima delle due, cioè **a=b**, ha per effetto che il valore di b (cioè 2) viene trascritto in a e quindi a questo punto sia a che b hanno il valore 2 e quando viene eseguita la seconda istruzione (cioè **b=a**) in b viene trascritto il valore *attuale* di a (cioè 2); il risultato finale sarà che comparirà sullo schermo la scritta

```
a=2     b=2
```

che non è assolutamente ciò che avremmo voluto vedervi comparire.

A nulla vale tentare di scambiare l'ordine delle due istruzioni, come nel seguente programma

Esempio 1.10

```
 /* QUESTO PROGRAMMA SCAMBIA I VALORI DI DUE VARIABILI
 IN MODO SBAGLIATO */
 #include <stdio.h>
 #include <stdlib.h>
  main()
  {
   int a,b;
   printf("\n\n\n valori di a e b prima dello scambio");
   printf( "\n\n     a=") ; scanf("%d",&a);
   printf( "\n     b=") ; scanf("%d",&b);
   b=a;a=b; // ECCO DOVE STA LO SBAGLIO
   printf("\n\n\n valori di a e b dopo lo scambio");
   printf("\n\n     a=%d    b=%d",a,b);
   system ("PAUSE");return 0;
  }
```

Il risultato di quest'ultimo programma sarà semplicemente che stavolta comparirà sullo schermo la scritta

```
 a=1    b=1
```

ma le due variabili non sono state scambiate neanche questa volta. Il problema, come ora capiamo, sta nel fatto che nel momento in cui viene eseguita, per esempio, l'istruzione **a=b**, il valore *attuale* di b viene trascritto in a e il valore che aveva a viene semplicemente *perduto*. Il modo *giusto* per effettuare uno scambio di valori fra due variabili, richiede quindi di utilizzare una terza variabile (*variabile di appoggio*) in cui andare a memorizzare momentaneamente il valore della variabile a che altrimenti andrebbe perso con l'istruzione **a=b**. Ecco un programma che esegue correttamente lo scambio di due variabili:

Esempio 1.11

```
 /* QUESTO PROGRAMMA SCAMBIA I VALORI DI DUE VARIABILI
 IN MODO CORRETTO */
 #include <stdio.h>
 #include <stdlib.h>
  main()
  {
   int a,b,appo;
   printf("\n\n\n valori di a e b prima dello scambio");
   printf( "\n\n     a=") ; scanf("%d",&a);
   printf( "\n     b=") ; scanf("%d",&b);
   appo=a;a=b;b=appo;
   printf("\n\n\n valori di a e b dopo lo scambio");
   printf("\n\n    a=%d    b=%d",a,b);
   system ("PAUSE");return 0;
  }
```

Come si vede, nella dichiarazione delle variabili, abbiamo introdotto anche una terza variabile **appo** (anch'essa di tipo **int**) che serve appunto per memorizzare *momentaneamente* il valore della variabile **a** in modo che il suo valore originario non vada perduto.

Capitolo 2

Istruzioni condizionali

In C++ l'istruzione condizionale nella sua forma più semplice ha la seguente struttura:

```
if (condizione) istruzione;
```

dove '**condizione**' è (di solito) una espressione logica che può essere vera o falsa, mentre '**istruzione**' è una qualsiasi istruzione eseguibile. Se '**condizione**' è soddisfatta (cioè è vera), allora '**istruzione**' viene eseguita, altrimenti si passa alla successiva istruzione del programma. Ecco come possiamo utilizzare il costrutto '**if**' per decidere quale è il più grande fra due numeri interi assegnati:

```
Esempio 2.1
// QUESTO PROGRAMMA DETERMINA IL MAGGIORE
// FRA DUE NUMERI a E b
 #include <stdio.h>
 #include <stdlib.h>
 main()
 {
   int a,b,max;
   printf("\n\n\n\nprimo numero a=");scanf("%d",&a);
   printf("\nsecondo numero b=");scanf("%d",&b);
   max=a;
   if (b>max) max=b;
   printf("\n\nil massimo fra %d e %d è %d",a,b,max);
   system ("PAUSE");return 0;
 }
```

Nelle prime tre righe del programma vengono dichiarate le tre variabili intere a,b e max e successivamente vengono *letti* (da tastiera) i valori di a e b; a questo punto la strategia per decidere quale è più grande fra a e b è la seguente: inizialmente mettiamo in max il valore di a, dopo di che, se il valore di b è maggiore del valore di max (e quindi del valore di a) allora cambiamo il valore di max con il valore di b, altrimenti lasciamo le cose come stanno. Il risultato è che alla fine in max ci sarà sicuramente il più grande fra i valori di a e b. Questa strategia per trovare il massimo fra due numeri dati può essere facilmente estesa per trovare il massimo fra un numero qualsiasi di numeri assegnati; ecco, per esempio, un programma che trova il massimo fra quattro numeri dati usando essenzialmente lo stesso trucco usato nell'Esempio 2.1:

Esempio 2.2

```
/* QUESTO PROGRAMMA DETERMINA IL MAGGIORE FRA
  QUATTRO NUMERI a,b,c E d */
#include <stdio.h>
#include <stdlib.h>
 main()
{
  int a,b,c,d,max;
  printf("\n\n\n\nprimo numero a=");scanf("%d",&a);
  printf("\nsecondo numero b=");scanf("%d",&b);
  printf("\nterzo numero c=");scanf("%d",&c);
  printf("\nquarto numero d=");scanf("%d",&d);
  max=a;
  if (b>max) max=b;if (c>max) max=c;if (d>max) max=d;
  printf("\n il massimo fra %d,%d,%d e %d è %d",a,b,c,d,max);
  system ("PAUSE");return 0;
}
```

Tutti sappiamo che una equazione di secondo grado del tipo

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

con coefficienti reali a, b e c, possiede due radici reali distinte se il discriminante $\Delta = b^2 - 4ac$ è positivo, altrimenti possiede una unica radice reale se il discriminante è nullo e infine possiede due radici complesse coniugate se il discriminante è negativo. Ecco un semplice programma in cui sfruttiamo di nuovo il costrutto **if** per decidere se una data equazione di secondo grado possiede due radici reali distinte oppure no:

Esempio 2.3

```
 /* QUESTO PROGRAMMA LEGGE I COEFFICIENTI a,b E c
   DI UN POLINOMIO DI SECONDO GRADO, NE CALCOLA IL
   DISCRIMINANTE E, SE QUESTO E' POSITIVO, SEGNALA
   CHE ESISTONO DUE RADICI REALI DISTINTE */
 #include <stdio.h>
 #include <stdlib.h>
  main()
  {  float a,b,c,delta;
     printf("\n\n\n\n\n");
     printf( "\nprimo coefficiente   a=");scanf("%f",&a);
     printf( "\nsecondo coefficiente b=");scanf("%f",&b);
     printf( "\nterzo coefficiente c=");scanf("%f",&c);
     printf("\n\n\n\n\n");
     delta=b*b-4*a*c;
     if (delta>0)
     printf("\nci sono due radici reali distinte\n");
    system ("PAUSE");return 0; }
```

In questo programma le variabili a, b, c e delta devono poter assumere valori reali; non sono più quindi variabili che possono assumere solo valori interi come nei precedenti esempi. Non sarà quindi possibile dichiararle mediante **int**, ma dovremo utilizzare la dichiarazione **float**, che serve appunto a dichiarare le variabili di tipo reale; dal punto di vista pratico, il compilatore individuerà quattro aree di memoria di dimensione opportuna (tipicamente di 4 bytes ciascuna) che da questo momento verranno riservate alle variabili a, b, c e delta rispettivamente. Si ponga attenzione anche al fatto che nelle istruzioni **scanf** con cui vengono lette a, b e c, si deve usare il formato **%f** (che è il formato per i numeri reali) anziché il formato **%d** (che è il formato per i numeri interi). Osserviamo che il risultato del programma dell'Esempio 2.3 non è del tutto simmetrico, infatti, se il discriminante è positivo compare sullo schermo la scritta

```
ci sono due radici reali distinte
```

altrimenti sullo schermo non compare nulla; se vogliamo rendere più simmetrico il risultato del programma, se cioè vogliamo far comparire comunque sullo schermo una scritta che ci dica se ci sono o non ci sono due radici reali distinte, possiamo utilizzare il costrutto **if** nella sua forma più generale, che è la seguente:

```
if (condizione) istruzione1;
else istruzione2;
```

se **condizione** è verificata, viene eseguita l'istruzione **istruzione1**, altrimenti viene eseguita l'istruzione **istruzione2**.

Ecco un programma in cui viene utilizzato il costrutto **if-else** per decidere se un dato trinomio possiede o non possiede due radici reali distinte:

Esempio 2.4

```
/* QUESTO PROGRAMMA LEGGE I COEFFICIENTI a,b E c DI UN
 POLINOMIO DI SECONDO GRADO, NE CALCOLA IL  DISCRIMINANTE
 E, SE QUESTO E' POSITIVO, SEGNALA CHE ESISTONO DUE
 RADICI REALI DISTINTE, ALTRIMENTI SEGNALA CHE NON
 CI SONO DUE RADICI REALI DISTINTE */
 #include <stdio.h>
 #include <stdlib.h>
  main()
 { float a,b,c,delta;
   printf("\n\n\n\n\n");
   printf( "\nprimo coefficiente   a=");scanf("%f",&a);
   printf( "\nsecondo coefficiente b=");scanf("%f",&b);
   printf( "\nterzo coefficiente c=");scanf("%f",&c);
   printf("\n\n\n\n\n");
   delta=b*b-4*a*c;
   if (delta>0)
      printf("\nci sono due radici reali distinte\n");
   else
      printf("\nnon ci sono due radici reali distinte\n");
   system ("PAUSE");return 0; }
```

L'uso del costrutto **if-else** nel programma dell'Esempio 2.4 non sembra però del tutto soddisfacente, infatti esso consente di distinguere due soli casi, cioè il caso $\Delta > 0$ e il caso $\Delta \leq 0$, mentre sarebbe più logico distinguere tre casi, cioè il caso $\Delta > 0$, il caso $\Delta = 0$ e il caso $\Delta < 0$. Per fortuna non è difficile utilizzare il costrutto **if-else** per distinguere fra tre casi, infatti basta usarlo due volte annidandolo dentro se stesso; ecco come sarà la forma generale dell'**if-else** per distinguere fra tre casi:

```
if (condizione1) istruzione1;
else if (condizione2) istruzione2;
     else istruzione3;
```

Se la **condizione1** è soddisfatta si esegue la **istruzione1**, altrimenti si entra in un nuovo **if-else** e si controlla se è soddisfatta la **condizione2**; se questa è soddisfatta si esegue la **istruzione2**, altrimenti si esegue la **istruzione3**.

Ecco come possiamo distinguere i tre casi del discriminante di un trinomio:

Esempio 2.5

```
/* QUESTO PROGRAMMA LEGGE I COEFFICIENTI a,b E c
DI UN POLINOMIO DI SECONDO GRADO, NE CALCOLA IL
DISCRIMINANTE E, A SECONDA CHE QUESTO SIA POSITIVO,
NULLO O NEGATIVO, SEGNALA CHE ESISTONO DUE RADICI
REALI DISTINTE, UNA UNICA RADICE REALE O NESSUNA
RADICE REALE */
#include <stdio.h>
#include <stdlib.h>
 main()
{ float a,b,c,delta;
  printf("\n\n\n\n\n");
  printf( "\nprimo coefficiente   a=");scanf("%f",&a);
  printf( "\nsecondo coefficiente b=");scanf("%f",&b);
  printf( "\nterzo coefficiente c=");scanf("%f",&c);
  printf("\n\n\n\n\n");
  delta=b*b-4*a*c;
  if (delta>0)
      printf("\nci sono due radici reali distinte\n");
  else if (delta==0)
          printf("\nc'è una unica radice reale\n");
       else
          printf("\nnon ci sono radici reali\n");
  system ("PAUSE");return 0; }
```

Si osservi che la **condizione2** (che sarebbe $\Delta = 0$), deve essere scritta in C++ nella forma

```
delta==0;
```

infatti, come abbiamo visto, il simbolo di '=' semplice, in C++ è un operatore di assegnazione e non un simbolo di relazione. Non è difficile capire che, se noi annidiamo due volte il costrutto **if-else** dentro se stesso, potremo distinguere fra quattro casi, se lo annidiamo tre volte potremo distinguere fra cinque casi, e così via. Tuttavia il costrutto **if-else** ha sempre un inconveniente, che è il

seguente: ogni volta che viene distinto un caso si può eseguire solamente una istruzione, che, negli esempi precedenti, è una istruzione **printf** mediante la quale viene stampato sullo schermo un messaggio. In effetti, sarebbe più ragionevole che, una volta stabilito in quale situazione siamo, si procedesse al calcolo completo delle radici; in altre parole, sarebbe meglio che il nostro programma facesse qualcosa del genere:

- *se* $\Delta > 0$ *allora vengono calcolate e stampate sullo schermo le due radici reali*
$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \quad e \quad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$
- *se* $\Delta = 0$ *allora viene calcolata e stampata sullo schermo la unica radice reale*
$$x_1 = \frac{-b}{2a}$$
- *se* $\Delta < 0$ *allora vengono calcolate e stampate sullo schermo le due radici complesse coniugate*
$$x_1 = \frac{-b}{2a} + i\frac{\sqrt{-\Delta}}{2a} \quad e \quad x_2 = \frac{-b}{2a} - i\frac{\sqrt{-\Delta}}{2a}$$

Per ottenere tutto ciò, è sufficiente racchiudere un certo numero di istruzioni fra le parentesi graffe aperta e chiusa; infatti in C++ ogni gruppo di istruzioni racchiuso fra **{** e **}** viene considerato come una unica istruzione; quindi il costrutto **if-else** che abbiamo usato nell'Esempio 2.5 si può usare nella forma più generale:

```
if (condizione1) {gruppo di istruzioni 1};
else if (condizione2) {gruppo di istruzioni 2};
     else {gruppo di istruzioni 3};
```

Ecco come possiamo scrivere un programma che risolve in maniera completa una equazione di secondo grado:

```
 Esempio 2.6
 /* QUESTO PROGRAMMA LEGGE I COEFFICIENTI a,b,c DI UN POLINOMIO
 DI SECONDO GRADO, NE CALCOLA LE RADICI E LE STAMPA, SIA NEL
 CASO DI RADICI REALI  CHE NEL CASO DI RADICI COMPLESSE */
 #include <stdio.h>
 #include <math.h>
 #include <stdlib.h>
 main()
{ float a,b,c,delta,x1,x2,x,alfa,beta,aa,s;
  printf( "\nprimo coefficiente   a=");scanf("%f",&a);
  printf( "\nsecondo coefficiente b=");scanf("%f",&b);
  printf( "\nterzo coefficiente c=");scanf("%f",&c);
  delta=b*b-4*a*c; aa=2*a;
  if (delta>0)
     {x1=(-b+sqrt(delta))/aa; x2=(-b-sqrt(delta))/aa;
      printf("\nci sono due radici reali distinte\n");
      printf("\nprima radice   =%f",x1);
      printf("\nseconda radice   =%f",x2); }
   else  if (delta==0)
           {x=-b/aa;
            printf("\nc'è una unica radice reale =%f",x); }
         else
           {alfa=-b/aa; s=sqrt(-delta); beta=s/aa;
            printf("\nci sono due radici complesse \n");
            printf("\nprima radice =%f+i%f",alfa,beta);
            printf("\nseconda radice =%f-i%f",alfa,beta); }
   system ("PAUSE");return 0; }
```

Per quanto riguarda quest'ultimo programma, si osservi che, per calcolare la radice quadrata del discriminante, abbiamo utilizzato la *funzione* **sqrt** che è appunto la funzione che calcola la radice quadrata in C++; poiché la definizione di questa funzione si trova nel file **math.h**, abbiamo dovuto includerlo mediante la direttiva

```
#include <math.h>
```

in testa al programma.

Capitolo 3

Cicli

Supponiamo di voler scrivere un programma che legge un numero intero n e poi calcola e stampa il valore di n!; l'idea che useremo per ottenere n! è la seguente: prima di tutto introduciamo una variabile intera in cui andare a mettere il risultato e che chiameremo **fatt** e a cui assegneremo inizialmente il valore 1; dopo di che sarà sufficiente ripetere l'istruzione

```
fatt=fatt*i;
```

con la variabile intera i che assume successivamente tutti i valori da 1 ad n; in ultimo avremo in **fatt** il valore richiesto di n!, per cui basterà stampare **fatt**. Per scrivere un programma in C++ che effettui questa sequenza di operazioni, occorre introdurre delle nuove istruzioni, che sono dette istruzioni di ciclo. La più semplice istruzione di questo tipo è l'istruzione **for** che, nei casi più comuni, si presenta al modo seguente

```
for (istruzione1;condizione;istruzione2) istruzione3;
```

dove la **istruzione1** (detta anche istruzione di inizializzazione) serve a fissare il valore iniziale della variabile intera con cui si compie il ciclo e nel caso del fattoriale dovrebbe quindi essere i=1; la **condizione** (detta anche condizione di iterazione) serve per decidere fino a quando il ciclo andrà ripetuto e nel caso del fattoriale dovrebbe quindi essere $i <= n$ (ricordiamo che in C++ la disuguaglianza $i \leq n$ si scrive $i <= n$); la **istruzione2** (detta anche istruzione di aggiornamento) serve ad aggiornare la variabile con cui si compie il ciclo e nel caso del fattoriale dovrebbe essere $i = i+1$; infine la **istruzione3** è l'istruzione che deve essere ripetuta per tutto il ciclo e che nel caso del fattoriale sarà

```
fatt=fatt*i.
```

(Si osservi che anche in questo caso l'istruzione che deve essere ripetuta per tutto il ciclo, può essere sostituita da un gruppo di istruzioni a patto che vengano racchiuse fra le parentesi graffe { e }).

Ecco come si presenterà un programma che calcola il fattoriale di un intero assegnato col procedimento sopra descritto:

```
Esempio 3.1
 /* QUESTO PROGRAMMA CALCOLA E STAMPA IL FATTORIALE DI
 UN NUMERO n FORNITO IN LETTURA */
  #include <stdio.h>
  #include <stdlib.h>
  main()
  {int i,n,fatt;
   printf("\n\n\n\n   n=");scanf("%d",&n);
   fatt=1;
   for (i=1;i<=n;i++) fatt=fatt*i;
   printf("\n\n\n\nfattoriale di %d",n);
   printf("=%d",fatt);
   system ("PAUSE");return 0; }
```

Osserviamo che l'istruzione di aggiornamento nel ciclo **for** è scritta nella forma

```
i++;
```

anziché nella forma

```
i=i+1;
```

questa *abbreviazione* è comunemente usata in C++ e in modo analogo sarà possibile usare l'*abbreviazione*

```
i--
```

al posto di

```
i=i-1.
```

A questo proposito possiamo osservare subito che n! può essere calcolato sia facendo il prodotto dei numeri interi da 1 ad n, sia facendo il prodotto dei numeri interi da n ad 1; ecco come dobbiamo modificare il programma precedente per calcolare n! moltiplicando i numeri interi a partire da n e scendendo fino ad 1:

Esempio 3.2 /* QUESTO PROGRAMMA CALCOLA E STAMPA IL FATTORIALE DI UN NUMERO n FORNITO IN LETTURA */

```
#include <stdio.h>
#include <stdlib.h>
main()
{
 int i,n,fatt;
 printf("\n\n\n\n   n=");scanf("%d",&n);
 fatt=1;
 for(i=n;i>=1;i--) fatt=fatt*i;
 printf("\n\n\n\nfattoriale di %d",n);
 printf("=%d",fatt);
 system ("PAUSE");return 0;
}
```

Se proviamo a compilare e ad eseguire più volte i due programmi precedenti assegnando ad n valori diversi, scopriremo rapidamente che, fintanto che n non è molto grande (e precisamente finché n non supera 12) il programma fornisce un risultato esatto; tuttavia per n=13, n=14, ecc. il programma sembra impazzire e fornisce valori assurdi (addirittura negativi talvolta). La ragione di questo strano comportamento è molto semplice: dal momento che la porzione di memoria riservata ad una variabile intera è *finita* (4 bytes), di conseguenza solo un numero *finito* di numeri interi vi sarà rappresentabile (per la precisione i numeri interi da -2147483648 fino a +2147483647). Poiché 13! vale 6227020800, esso eccede il limite di rappresentabilità degli interi; purtroppo il compilatore C++ non si accorge di ciò, infatti esso, quando raggiunge il valore limite +2147483647, prosegue semplicemente ricominciando dal numero -2147483648 (ecco perché si possono ottenere valori negativi!). Quando trattiamo con numeri interi, dobbiamo quindi sempre tenere conto di questa limitazione del compilatore C++ per evitare di trovarci in situazioni simili. Ecco un *trucco* che possiamo usare per calcolare il fattoriale di numeri interi più grandi di 12:

```
Esempio 3.3
 /* QUESTO PROGRAMMA CALCOLA E STAMPA
 IL FATTORIALE DI UN NUMERO n FORNITO IN LETTURA */
#include <stdio.h>
#include <stdlib.h>
 main()
{int i,n;float fatt;
 printf("\n\n\n\n   n=");scanf("%d",&n);
 fatt=1;
 for(i=1;i<=n;i++) fatt=fatt*i;
 printf("\n\n\n\nfattoriale di %d",n);
 printf("=%f",fatt);
 system ("PAUSE");return 0; }
```

Come si vede, in quest'ultimo programma abbiamo semplicemente dichiarato che **fatt** sia di tipo **float**, cioè un numero reale; dal punto di vista delle operazioni ciò non crea nessun inconveniente, infatti, quando si tratta di moltiplicare **fatt** (che è un **float**) per i (che è un **int**), il compilatore C++ esegue l'operazione senza problemi e fornisce un risultato che è ancora un **float**. In questo modo possiamo sfruttare il fatto che il massimo numero rappresentabile di tipo **float** è molto più grande del massimo numero rappresentabile di tipo **int**. Se tutto ciò non bastasse, abbiamo ancora altre frecce al nostro arco; per esempio esistono le variabili reali di tipo **double** che hanno a disposizione uno spazio di memoria più grande delle variabili di tipo **float**.

In tutti i programmi che abbiamo scritto finora per calcolare il fattoriale di un numero intero, abbiamo sempre usato l'istruzione **for** per eseguire il ciclo di calcolo; tuttavia questa non è l'unica istruzione del C++ che può essere usata per eseguire un ciclo. Un'altra istruzione che si usa per eseguire cicli è l'istruzione **while**, che ha la seguente sintassi:

```
while (condizione) istruzione;
```

Se **condizione** è vera, allora viene eseguita **istruzione** e successivamente si ritorna a valutare **condizione**, se questa è ancora vera si esegue di nuovo **istruzione** e così via fino a che **condizione** risulta falsa, nel qual caso si salta **istruzione** e si prosegue il programma. Un'altra istruzione imparentata con **while** e che si può usare per eseguire cicli è l'istruzione **do-while**, che ha la seguente sintassi:

```
do istruzione while (condizione);
```

La differenza rispetto al semplice **while** è che con il **do-while** prima si esegue **istruzione** e poi si valuta **condizione** e se questa è vera si torna ad eseguire **istruzione** altrimenti si prosegue nel programma; di conseguenza, **istruzione** viene eseguita comunque almeno una volta se si usa il **do-while**, mentre, con il semplice **while**, **istruzione** potrebbe non essere mai eseguita (se cioè fin dalla prima volta **condizione** risultasse falsa). Naturalmente, sia per quanto riguarda **while** che **do-while** si potrà sostituire **istruzione** con un gruppo di istruzioni racchiudendole entro le parentesi graffe, per cui la forma più generale dell'istruzione **while** sarà

```
while (condizione) {gruppo di istruzioni};
```

mentre la forma più generale dell'istruzione **do-while** sarà

```
do {gruppo di istruzioni} while (condizione);
```

Ecco qui di seguito due programmi con cui viene calcolato il fattoriale di un numero intero utilizzando appunto **while** e **do-while**

```
Esempio 3.4
/* QUESTO PROGRAMMA CALCOLA E STAMPA IL FATTORIALE
 DI UN NUMERO n FORNITO IN LETTURA */
 #include <stdio.h>
 #include <stdlib.h>
 main()
 { int i,n;int fatt;
   printf("\n\n\n\n   n=");scanf("%d",&n);
   fatt=1;i=1;
   while (i<=n) { fatt=fatt*i;i=i+1;}
   printf("\n\n\n\nfattoriale di %d=%d\n\n",n,fatt);
   system ("PAUSE");return 0; }
```

```
Esempio 3.5 /* QUESTO PROGRAMMA CALCOLA E STAMPA IL FATTORIALE DI
 UN NUMERO n FORNITO IN LETTURA */
 #include <stdio.h>
 #include <stdlib.h>
 main()
 { int i,n;int fatt;
   printf("\n\n\n\n   n=");scanf("%d",&n);
   fatt=1;i=1;
   do {i++;fatt=fatt*i;} while (i<n);
   printf("\n\nfattoriale di %d",n);
   printf("=%d",fatt);
   system ("PAUSE");return 0;  }
```

Si osservi che la **condizione** negli Esempi 3.4 e 3.5 è diversa appunto perché nel primo caso si usa il **while** semplice e quindi bisogna richiedere la condizione $i \leq n$, mentre nel secondo caso si usa il **do-while** e quindi bisogna richiedere la condizione $i < n$.

Il trucco che abbiamo usato per calcolare il prodotto dei numeri interi da 1 ad n, può essere facilmente modificato per calcolare, anziché il prodotto di n numeri, la somma di n numeri; per esempio, immaginiamo di voler scrivere un programma che calcola la media di n numeri reali: prima di tutto il nostro programma dovrà *leggere* il valore di una variabile n di tipo **int**, successivamente definiremo una variabile di tipo **float** che chiameremo **media** e a cui assegneremo inizialmente il valore 0, a questo punto inizieremo un *ciclo* che consisterà nel ripetere per n volte le due seguenti operazioni:

1. leggi il valore di una variabile x di tipo **float**
2. esegui la somma **media=media+x**

Al termine del ciclo, nella variabile media sarà memorizzato il valore della somma degli n numeri reali letti; a questo punto sarà sufficiente aggiungere l'istruzione **media=media/n** e poi far stampare in uscita il valore di **media** .

Ecco come si presenta un programma in C++ che calcola il valor medio di n numeri assegnati tramite la tastiera:

```
Esempio 3.6 /* QUESTO PROGRAMMA CALCOLA E STAMPA IL VALOR MEDIO DI
   n NUMERI REALI FORNITI IN LETTURA */
   #include <stdio.h>
   #include <stdlib.h>
   main()
   { int n,i;float x;
     printf("\n\n\n\nquanti sono i numeri");
     printf(" di cui devo calcolare la media?");
     printf("\n  n= ");scanf("%d",&n);
     float media=0;
     for(i=1;i<=n;i++)
        { printf("\n dammi un numero reale x=");
          scanf("%f",&x);
          media=media+x; }
     media=media/n;
     printf("\n\n\n\n  ecco il valor medio: %f",media);
     system ("PAUSE");return 0; }
```

Negli esempi precedenti all'Esempio 3.6, avevamo posto sempre all'inizio del programma le istruzioni in cui si dichiarava il tipo delle variabili usate nel seguito; ciò non è tuttavia strettamente obbligatorio, e la dichiarazione di tipo di una variabile può essere posta in un punto qualsiasi del programma purché prima che la variabile in questione venga utilizzata la prima volta. Nell'Esempio 3.6, abbiamo utilizzato appunto questa possibilità definendo il tipo della variabile **media** solo nel momento in cui la abbiamo utilizzata; si osservi a questo proposito che l'istruzione

```
float media=0;
```

costituisce una abbreviazione delle due seguenti istruzioni:

```
 float media;media=0;
```

Il ciclo del programma dell'Esempio 3.6 può essere facilmente scritto utilizzando le istruzioni di ciclo **do-while** o **while**; lasciamo ciò al lettore come utile esercizio.

Capitolo 4

Array

In C++ un array ad una dimensione è semplicemente una lista di variabili che hanno tutte lo stesso nome e che vengono distinte una dall'altra mediante un indice intero; per esempio, se a è un array di tipo **int**, esso è un insieme (finito) di variabili di tipo **int** la prima delle quali viene identificata da a[0], la seconda da a[1], la terza da a[2],...,la n-esima da a[n-1], ecc.; quando usiamo un array in un programma C++, dobbiamo dichiararlo (esattamente come facciamo per le altre variabili), e nella dichiarazione dovrà essere specificato

1. il tipo delle variabili che formano l'array
2. il nome comune a tutte le variabili che formano l'array
3. quante sono le variabili che formano l'array

e quindi la dichiarazione **int a[10];** serve a definire un array formato da 10 variabili intere di tipo **int** che verranno identificate una per una da a[0],a[1],a[2],...,a[9] e analogamente la dichiarazione **float v[7];** serve a definire un array formato da 7 variabili reali di tipo **float** che verranno identificate una per una da v[0],v[1],v[2],...,v[6]. Come si vede quindi la prima variabile di un array ha sempre l'indice 0 e la n-esima variabile di un array avrà quindi sempre indice n-1; questo fatto è talvolta un po' scomodo, ma, come vedremo in seguito, presenta anche alcuni vantaggi non indifferenti.

Tra gli *oggetti* matematici che più comunemente vengono rappresentati utilizzando un array, ci sono i vettori; sappiamo che, per esempio, un vettore dell'ordinario spazio euclideo (reale) ad n dimensioni può essere identificato con una n-pla di numeri reali e pertanto appare naturale rappresentare in C++ un vettore ad n dimensioni mediante un array che abbia (almeno) n elementi. Ecco per esempio un programma che calcola la media di n numeri assegnati e nel quale noi memorizziamo gli n numeri come le componenti di un vettore x:

Esempio 4.1

```
 /* QUESTO PROGRAMMA CALCOLA E STAMPA IL VALOR MEDIO DI
 n NUMERI REALI FORNITI IN LETTURA COME COMPONENTI DI UN VETTORE */
 #include <stdio.h>
 #include <stdlib.h>
 main()
 {int n,i;float media;float x[20];
  printf("\n\n\n\nquante componenti ha il vettore");
  printf(" che devo leggere?");
  printf("\n  n= ");scanf("%d",&n);
  if (n>20) return 0;
  for(i=1;i<=n;i++)
    { printf("\n dammi la componente numero");
      printf(" %d del vettore x ",i);
      scanf("%f",&x[i-1]); }
  media=0;
  for(i=0;i<n;i++) media=media+x[i];
  media=media/n;
  printf("\n\n\n\n  ecco il valor medio: %f",media);
  system ("PAUSE");return 0; }
```

Facciamo alcune osservazioni sul programma dell'Esempio 4.1:

- la dichiarazione

  ```
  float x[20];
  ```

  avverte il compilatore che l'array x è costituito da 20 variabili di tipo **float**, pertanto la dimensione n del vettore che leggeremo non potrà eccedere 20, altrimenti il nostro array non avrebbe lo spazio sufficiente per memorizzare tutte le componenti del vettore x. Ecco perché, subito dopo la lettura del valore di n abbiamo posto la istruzione

  ```
  if (n>20) return 0;
  ```

  il cui scopo è quello di interrompere l'esecuzione del programma nel caso appunto in cui sia $n > 20$. Purtroppo il compilatore C++, nel caso in cui si cercasse di memorizzare in un array più numeri di quelli previsti dalla dimensione dell'array, non segnalerebbe alcun errore; pertanto il controllo mediante l'istruzione

```
if (n>20) return 0;
```

è indispensabile per evitare di trovarsi in situazioni imprevedibili.

- nel ciclo **for**, l'indice i viene fatto variare da 1 ad n, ma per ogni valore di i viene letta la variabile x[i-1], ciò vuol dire che la i-esima componente del vettore che stiamo leggendo viene memorizzata nella variabile dell'array che ha nome x[i-1], e ciò dipende dal fatto che, come abbiamo detto prima, nell'array x gli indici partono dal valore 0.

- il programma dell'Esempio 4.1 calcola la media di n numeri dati, esattamente come il programma dell'Esempio 3.6, tuttavia c'è da sottolineare una importante differenza fra i due, che è la seguente: nel programma dell'Esempio 3.6 i numeri che vengono forniti in lettura sono tutti memorizzati nella medesima variabile x, e quindi ogni volta che viene letto un nuovo numero, il precedente viene *perduto*, cioè alla fine della esecuzione del programma l'unico numero di cui rimanga *memoria* è l'ultimo numero letto; viceversa nel programma dell'Esempio 4.1 vengono prima *memorizzati* nell'array x tutti gli n numeri forniti dall'utente attraverso la tastiera, e successivamente ne viene calcolata e stampata la media, pertanto fino alla fine del programma sono *accessibili* tutti gli n numeri letti. (Si osservi anche che, per ragioni di chiarezza abbiamo utilizzato due cicli **for**, uno per la lettura dei dati e l'altro per calcolare la media, tuttavia avremmo potuto facilmente usare un unico ciclo **for** come nell'Esempio 3.6; lasciamo al lettore il compito di effettuare per esercizio la necessaria modifica)

Nel programma dell' Esempio 4.1 l'istruzione

```
if (n>20) return 0;
```

ha l'effetto di interrompere il programma nel caso in cui n risulti maggiore di 20, tuttavia, per maggiore chiarezza, sarebbe meglio se, nel caso in cui fosse $n > 20$, venisse anche stampato un messaggio di errore; ecco come possiamo facilmente modificare il programma precedente per ottenere ciò:

Esempio 4.2

```
 /* QUESTO PROGRAMMA CALCOLA E STAMPA IL VALOR MEDIO DI
   n NUMERI REALI FORNITI IN LETTURA COME COMPONENTI
  DI UN VETTORE */
 #include <stdio.h>
 #include <stdlib.h>
 main()
 { int n,i;float media;float x[20];
   printf("\n\n\n\nquante componenti ha il vettore");
   printf(" che devo leggere?");
   printf("\n  n= ");scanf("%d",&n);
   if (n>20) {printf("\n\n\n attenzione!!");
              printf("\n ci sono troppe componenti!!");
              system("PAUSE");return 0; }
   for(i=1;i<=n;i++)
      { printf("\n dammi la componente numero");
        printf(" %d del vettore x ",i);
        scanf("%f",&x[i-1]); }
   media=0;for(i=0;i<n;i++) media=media+x[i];media=media/n;
   printf("\n\n\n\n  ecco il valor medio: %f",media);
   system ("PAUSE");return 0; }
```

Nei programmi dell'Esempio 4.1 e dell'Esempio 4.2, l'esecuzione si interrompe nel caso in cui sia $n > 20$; sarebbe meglio fare in modo invece che, se $n > 20$, il programma rifiutasse il valore assegnato stampando sullo schermo un opportuno messaggio di errore, ma che poi continuasse l'esecuzione richiedendo un nuovo valore in lettura per n; ecco come possiamo ottenere tutto ciò attraverso l'uso del costrutto **while**:

Esempio 4.3

```
 /* QUESTO PROGRAMMA CALCOLA E STAMPA IL VALOR MEDIO
  DI n NUMERI REALI FORNITI IN LETTURA COME
  COMPONENTI DI UN VETTORE */
  #include <stdio.h>
  #include <stdlib.h>
  main()
  {int n,i;float media;
   float x[20];
   printf("\n\n\n\nquante componenti ha il vettore");
   printf(" che devo leggere?");
   printf("\n  n= ");scanf("%d",&n);
   while (n>20)
       {printf("\n\n\n attenzione!!");
        printf("\n le componenti del vettore sono troppe");
        printf("\nscegli di nuovo quante componenti deve");
        printf(" avere il vettore che devo leggere");
        printf("\n  n= ");scanf("%d",&n); }
  for(i=1;i<=n;i++)
     {printf("\n dammi la componente numero");
      printf(" %d del vettore x ",i);
      scanf("%f",&x[i-1]); }
  media=0;
  for(i=0;i<n;i++) media=media+x[i];
  media=media/n;
  printf("\n\n\n\n  ecco il valor medio: %f",media);
  system ("PAUSE");return 0; }
```

Nel programma dell'Esempio 2.2 avevamo visto come fosse possibile trovare il massimo fra quattro numeri assegnati; poiché il trucco che avevamo usato per fare ciò risulta facilmente estensibile ad un numero n qualsiasi di numeri assegnati, possiamo facilmente scrivere un programma che legge un vettore ad n componenti e ne calcola la componente massima.

Ecco come potrebbe essere un programma del genere:

Esempio 4.4

```
 /* QUESTO PROGRAMMA TROVA LA MASSIMA COMPONENTE DI UN
   VETTORE FORNITO IN LETTURA */
 #include <stdio.h>
 #include <stdlib.h>
 main()
 {int n,i;float max;
  float v[31];
  printf("\n\n\n\nquante componenti ha il vettore");
  printf(" che devo leggere?");
  printf("\n  n= ");scanf("%d",&n);
  while (n>30)
    { printf("\n\n\n attenzione!!");
      printf("\n le componenti del vettore sono troppe");
      printf("\nscegli di nuovo quante componenti deve");
      printf(" avere il vettore che devo leggere");
      printf("\n  n= ");scanf("%d",&n); }
  for(i=1;i<=n;i++)
    { printf("\n dammi la componente numero");
      printf(" %d del vettore x ",i);
      scanf("%f",&v[i]); }
  max=v[1];
  for(i=2;i<=n;i++) if (max<v[i]) max=v[i];
  printf("\n\n\n\n massima componente=%f",max);
  system ("PAUSE");return 0; }
```

Per quanto riguarda quest'ultimo programma, osserviamo che, nel memorizzare il vettore nell'array v, abbiamo scelto di memorizzare la componente i-esima del vettore nella variabile v[i] (anziché nella variabile v[i-1] come in precedenza); ciò comporta due conseguenze:

1. la variabile v[0] non viene utilizzata mai per memorizzarvi alcuna componente del vettore

2. anche se l'array v è costituito da 31 variabili di tipo **float**, noi possiamo leggere vettori di dimensione al massimo 30 (appunto per il fatto che non utilizziamo la prima variabile dell'array) Tutto ciò ci porta ad evidenziare un fatto importante riguardo ai vettori ed agli array, che è il seguente:

un vettore ed un array sono due cose distinte.

Un vettore ad n componenti, cioè una n-pla di numeri, può venire memorizzato in un array, ma non necessariamente deve occuparlo tutto, anzi di solito non lo occupa affatto tutto ma ne occupa solo una parte. Chi scrive un programma deve quindi porre particolare attenzione a questo fatto, avendo sempre presente in quale *porzione* dell'array si trova un certo vettore. Nel programma dell'Esempio 4.4 abbiamo visto come trovare la massima componente di un vettore assegnato; se analizziamo l'algoritmo usato per ottenere ciò, appare subito evidente che, dopo l'esecuzione del ciclo

```
for(i=2;i<=n;i++) if (max<v[i]) max=v[i];
```

nella variabile **max** viene a trovarsi necessariamente il valore della massima componente del vettore assegnato, tuttavia non è possibile sapere quale fosse stato il valore dell'indice i della componente massima del vettore. Se noi volessimo scrivere un programma che, oltre a determinare e stampare la massima componente di un vettore, determina e stampa anche il valore dell'indice per cui viene conseguito il massimo, dovremmo introdurre anche una variabile **imax** (stavolta di tipo **int**) in cui andare a memorizzare il valore dell'indice i ogni volta che viene cambiato il valore di **max**.

Ecco come potremmo modificare il precedente programma per ottenere questo risultato:

Esempio 4.5

```
 /* QUESTO PROGRAMMA TROVA LA MASSIMA COMPONENTE DI UN
   VETTORE FORNITO IN LETTURA */
 #include <stdio.h>
 #include <stdlib.h>
 main()
 {int n,i,imax;float max;
  float v[31];
  printf("\n\n\n\nquante componenti ha il vettore");
  printf(" che devo leggere?");
  printf("\n  n= ");scanf("%d",&n);
  while (n>30)
    { printf("\n\n\n attenzione!!");
      printf("\n le componenti del vettore sono troppe");
      printf("\nscegli di nuovo quante componenti deve");
      printf(" avere il vettore che devo leggere");
      printf("\n  n= ");scanf("%d",&n); }
  for(i=1;i<=n;i++)
    { printf("\n dammi la componente numero");
      printf(" %d del vettore x ",i);
      scanf("%f",&v[i]); }
  max=v[1];imax=1;
  for(i=2;i<=n;i++) if (max<v[i])
              { max=v[i];
                imax=i;  }
  printf("\n\n\n\n massima componente=%f",max);
  printf("\n indice della massima componente=%d",imax);
  system ("PAUSE");return 0; }
```

Così come esistono gli array ad una dimensione, esistono in C++ anche gli array a due dimensioni, che sono semplicemente array con due indici anziché uno solo; le osservazioni che abbiamo fatto a proposito degli array ad una dimensione si possono estendere, con qualche ovvio aggiustamento, agli array a due dimensioni, quindi per esempio la dichiarazione

```
  float a[10][12];
```

serve a definire un array formato da 120(=10x12) variabili reali di tipo float che verranno identificate una per una da

```
a[0][0],a[0][1],a[0][2],...,a[0][11],
a[1][0],a[1][1],a[1][2],...,a[1][11],
.....................................
a[9][0],a[9][1],a[9][2],....,a[9][11]
```

Un uso ovvio degli array a due dimensioni in problemi matematici, è quello di utilizzarli per memorizzarvi una matrice; ecco qui di seguito un esempio di programma che fa le seguenti cose:

1. legge il numero delle righe

   `num_rig`

   e il numero delle colonne

   `num_col`

   di una matrice che dovrà essere introdotta in lettura tramite la tastiera

2. legge uno per uno gli elementi di una matrice delle dimensioni assegnate al punto 1)

3. calcola e stampa la somma degli elementi di ciascuna riga

```
Esempio 4.6
 #include <stdio.h>
 #include <stdlib.h>
 int main()
 {float a[20][20];float b[20];
  int num_rig,num_col,i,j;
  printf( "numero delle righe della matrice:    m=");
  scanf("%d",&num_rig);
  printf( "numero delle colonne della matrice:  n=");
  scanf("%d",&num_col);
  printf("\nintroduci gli elementi della matrice\n\n");
  for (i=1;i<=num_rig;i++)
     { for (j=1;j<=num_col;j++)
       { printf("\na[%d,%d]=",i,j);
         scanf("%f",&a[i][j]); } }
  for (i=1;i<=num_rig;i++)
     { b[i]=0;
       for (j=1;j<=num_col;j++)
       b[i]=b[i]+a[i][j]; }
  for (i=1;i<=num_rig;i++)
     printf("\nb(%d)=%f",i,b[i]);
  system ("PAUSE");return 0; }
```

Si osservi che nell'Esempio 4.6 abbiamo utilizzato sia un array a due dimensioni a (per memorizzare la matrice), sia un array ad una dimensione b (per memorizzare le somme degli elementi di ciascuna riga della matrice); inoltre, sia per effettuare la lettura della matrice elemento per elemento, sia per calcolare le somme degli elementi di ciascuna riga, abbiamo dovuto utilizzare un doppio ciclo for, nel senso che abbiamo dovuto inserire due volte un ciclo for dentro un altro ciclo for.

Capitolo 5

Funzioni

Cominciamo con l'esaminare il seguente programma C++:

```
Esempio 5.1
// QUESTO PROGRAMMA CALCOLA IL VALORE ASSOLUTO
// (O MODULO) DI UN NUMERO REALE ASSEGNATO
 #include <math.h>
 #include <iostream>
 using namespace std;
  main()
 {float x,y;
  cout<<"\ndammi un numero reale x=";
  cin>>x;
  y=fabs(x);
  cout<<"\n\n modulo di  ";
  cout<<x;
  cout<<" = ";
  cout<<y;
  system("PAUSE");return 0; }
```

Rispetto ai programmi che abbiamo visto in precedenza, ci sono alcune novità che ora commenteremo: prima di tutto possiamo vedere che le istruzioni di ingresso e uscita dei dati non sono più **scanf** e **printf**; per far entrare ed uscire i dati vengono ora utilizzate **cin**>> e **cout**<<, che sembrano anche più semplici da usare rispetto a **scanf** e **printf**, infatti **cin**>> richiede semplicemente che gli venga accodato il nome della variabile che si vuole introdurre (da tastiera), senza bisogno di specificare il suo tipo come si faceva con **scanf**, e analogamente **cout**<< richiede che gli venga accodato il nome della variabile o la sequenza di caratteri (racchiusa fra le virgolette)

che si vuole stampare sul video. Si osservi che anche per **cout**<< è possibile (come per **printf**) compattare diverse istruzioni successive di stampa; ecco come si presenterà il programma dell' Esempio 5.1 compattando le istruzioni **cout**<<:

```
Esempio 5.2
// QUESTO PROGRAMMA CALCOLA IL VALORE ASSOLUTO
// (O MODULO) DI UN NUMERO REALE ASSEGNATO
 #include <math.h>
 #include <iostream>
 using namespace std;
 main()
 {float x,y;
  cout<<"\ndammi un numero reale x=";
  cin>>x;
  y=fabs(x);
  cout<<"\n\n modulo di  "<<x<<" = "<<y;
  system("PAUSE");return 0; }
```

Come si vede, non occorre ripetere **cout**<< ogni volta, ma è sufficiente accodare alla precedente variabile (o sequenza di caratteri) in uscita un nuovo simbolo << e la successiva variabile (o sequenza di caratteri) che si vuole mandare in uscita subito dopo.

E' importante osservare che anche **cin**>> e **cout**<< (come **scanf** e **printf**) sono definiti in un apposito file della Libreria Standard, che è il file **iostream** (nel caso di **scanf** e **printf** era invece il file **stdio.h**); ecco perché in testa al programma abbiamo dovuto questa volta aggiungere

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

Un utile esercizio potrebbe essere quello di riscrivere tutti i programmi che abbiamo esaminato fino ad adesso usando **cin**>> e **cout**<< al posto di **scanf** e **printf**.

Passiamo ora a commentare l'istruzione

```
 y=fabs(x);
```

in cui viene utilizzata la funzione **fabs**, che è una funzione matematica che calcola il valore assoluto (o modulo) di un numero reale (per calcolare invece il modulo di un numero intero si usa la funzione **abs**); per calcolare questa particolare funzione, il compilatore deve consultare il file **math.h** (dove questa funzione è definita), e quindi abbiamo dovuto premettere al programma la direttiva

```
#include <math.h>
```

In un precedente programma avevamo dovuto usare questa stessa direttiva per poter utilizzare la funzione matematica sqrt (che calcola la radice quadrata di un numero reale); è ben evidente quindi che nel file **math.h** si troveranno molte altre funzioni matematiche, per esempio potremo trovarvi le funzioni sin, cos e tan che calcolano rispettivamente il seno, il coseno e la tangente trigonometrici di un numero reale, oppure potremo trovarvi le funzioni log, log10 ed exp che calcolano rispettivamente il logaritmo naturale, il logaritmo in base 10 e l'esponenziale di un numero reale, e così via molte altre funzioni di uso comune in matematica. L'uso di tutte queste funzioni matematiche è abbastanza intuitivo e abbastanza semplice: per esempio, nel caso della istruzione **y=fabs(x);** verrà preso il valore attuale della variabile x (che è quello introdotto, tramite tastiera, dall'istruzione **cin>>x;**), ne verrà calcolato il valore assoluto (**fabs(x)**) e il numero così ottenuto verrà assegnato come valore alla variabile y. Ciò che non è invece affatto intuitivo è: che cosa il compilatore trova nel file **math.h**? E' ovvio che il compilatore troverà in **math.h** le istruzioni per calcolare le varie funzioni matematiche, ma il punto che noi vorremmo chiarire è il seguente: come sono fatte queste istruzioni? Per rispondere a queste domande, mostreremo ora un programma equivalente a quello dell'Esempio 5.2, ma nel quale la funzione che calcola il modulo di un numero reale è definita direttamente da noi, senza dover ricorrere al file **math.h**. Ecco come potrebbe essere fatto un programma del genere:

```
Esempio 5.3
// QUESTO PROGRAMMA CALCOLA IL VALORE ASSOLUTO
// (O MODULO) DI UN NUMERO REALE ASSEGNATO
 #include <iostream>
 using namespace std;
 float modulo(float);
 main()
 {float x,y;
  cout << "\ndammi un numero reale x=";cin >> x;
  y=modulo(x);
  cout << "\n\n modulo di  " << x << " = " << y;
  system("PAUSE");return 0; }
 float modulo(float a)
 {if (a>=0) return a;
  else return -a; }
```

Prima di tutto osserviamo che la funzione che calcola il valore assoluto è stata chiamata **modulo** anziché **fabs**; dal momento che la funzione **modulo**

viene definita esplicitamente da noi, possiamo darle il nome che vogliamo (con qualche ragionevole limitazione di cui parleremo in seguito). La definizione vera e propria della funzione **modulo** è costituita dalle ultime cinque righe, cioè:

```
  float modulo(float a)
 {if (a>=0) return a;
  else return -a; }
```

La prima di queste righe (detta testata della funzione), cioè

```
float modulo(float a)
```

serve a specificare tre cose:

1. che tipo di risultato fornisce la funzione (**float**)
2. il nome della funzione (**modulo**)
3. il tipo e il nome dell'argomento (**float a**)

Dopo la testata della funzione seguono due parentesi graffe **{** e **}** entro le quali si trova il corpo della funzione, cioè le istruzioni

```
 if (a>=0) return a;
  else return -a;
```

che sono appunto le istruzioni che servono a specificare che cosa deve calcolare la funzione **modulo**. Il significato di queste due ultime istruzioni dovrebbe essere abbastanza chiaro: quando il valore di a è maggiore o uguale a zero, la funzione modulo restituisce al **main** il valore stesso di a (**return a;**), altrimenti se a è minore di zero restituisce il valore di a cambiato dei segno (**return -a;**), il che corrisponde esattamente a calcolare il valore assoluto di a. Le istruzioni di ritorno (cioè **return a;** e **return -a;**) hanno lo scopo sia di restituire un valore al programma chiamante (cioè al **main**), sia di terminare l'esecuzione della funzione **modulo** e di ritornare all'esecuzione del programma chiamante (cioè al **main**).

Tutto ciò sembra abbastanza ragionevole e logico, tuttavia qualcuno si potrebbe porre la seguente domanda: perché nel **main** la funzione **modulo** viene applicata alla variabile x, mentre nella funzione **modulo** compare la variabile a? La risposta è questa: la variabile a è una variabile *locale* nella funzione **modulo**. Quando il programma principale (cioè il **main**) incontra la funzione **modulo** applicata all'argomento x, il valore che ha la variabile x nel **main** viene passato come valore iniziale alla variabile *locale* a della

funzione **modulo** e vengono eseguite le istruzioni previste dalla funzione **modulo** fino a che questa restituisce (tramite **return a** oppure **return -a)** al **main** il valore di **modulo(x)** (da assegnare alla variabile y). In effetti la variabile *locale* a della funzione **modulo** esiste fisicamente solo durante l'esecuzione della funzione **modulo** e termina di esistere nel momento in cui si ritorna all'esecuzione del programma chiamante (cioè, in questo caso, il **main**).

Si osservi anche che, dove prima compariva la direttiva

```
#include <math.h>
```

(che è stata soppressa perché non serve più) ora compare l'istruzione

```
float modulo(float);
```

Quest'ultima è chiamata il *prototipo* della funzione **modulo**, e serve ad avvertire il compilatore che la funzione **modulo** è una funzione che sarà definita più avanti (nel nostro caso dopo il **main {.....}**). Si osservi che quando si dichiara un prototipo di funzione, non occorre specificare il nome dell'argomento della funzione, ma solo il tipo cui appartiene il risultato della funzione (nel nostro caso il **float** che precede **modulo** indica che il risultato della funzione **modulo** sarà un numero reale) e il tipo dell'argomento (il **float** che segue **modulo** e che si trova tra due parentesi indica che anche l'argomento della funzione sarà un numero reale).

Riassumendo ciò che abbiamo imparato dall'esempio precedente, possiamo dire che, volendo definire una funzione da usare nel programma principale, un modo di procedere potrebbe essere il seguente (vedremo poi che ci sono altri modi possibili): anteporre al **main** il prototipo della funzione che vogliamo definire, che avrà la seguente struttura: tipo della funzione nome della funzione(tipo degli argomenti); posporre al **main** la testata della funzione che vogliamo definire, che avrà la seguente struttura

tipo della funzione nome della funzione(tipo e nome degli argomenti)

e subito di seguito il corpo della funzione che vogliamo definire racchiuso fra le parentesi graffe aperta e chiusa.

Nel caso specifico dell'esempio precedente, la funzione **modulo** ha un unico argomento, tuttavia è ovvio che sarà possibile definire funzioni che hanno più di un argomento; ecco un esempio di programma in cui viene definita una funzione norma di due argomenti che calcola la radice quadrata della somma dei quadrati di due numeri dati:

```
Esempio 5.4
//QUESTO PROGRAMMA CALCOLA LA RADICE QUADRATA DELLA SOMMA
//DEI QUADRATI DI DUE NUMERI X ED Y ASSEGNATI
 #include <math.h>
 #include <iostream>
 using namespace std;
  float norma(float,float);
  main()
 { float x,y,z;
   cout << "\ndammi un numero reale x=";cin >> x;
   cout << "\ndammi un numero reale y=";cin >> y;
   z=norma(x,y);
   cout << "\n\nla norma di " << x << " e " << y << " è " <<z;
   system("PAUSE");return 0; }
  float norma(float a,float b)
 { float c=sqrt(a*a+b*b);
   return c; }
```

Un'altra cosa che ovviamente è possibile fare è quella di definire e utilizzare più di una funzione all'interno di uno stesso programma; ecco, per esempio, come possiamo modificare il programma dell' Esempio 5.4 definendo sia la funzione **norma** che la funzione **quad** (che serve a calcolare il quadrato di un numero):

```
Esempio 5.5
//QUESTO PROGRAMMA CALCOLA LA RADICE QUADRATA DELLA SOMMA
//DEI QUADRATI DI DUE NUMERI X ED Y ASSEGNATI
 #include <math.h>
 #include <iostream>
 using namespace std;
  float norma(float,float);
  float quad(float);
  main()
 {float x,y,z;
  cout << "\ndammi un numero reale x=";cin >> x;
  cout << "\ndammi un numero reale y=";cin >> y;
  z=norma(x,y);
  cout << "\n\nla norma di " << x << " e " << y << " è " <<z;
  system("PAUSE");return 0; }
  float norma(float a,float b)
 {float c;
  c=sqrt(quad(a)+quad(b));
  return c; }
  float quad(float a)
 {float b;
  b=a*a;
  return b; }
```

In quest'ultimo programma è importante osservare che le variabili locali a e b nella funzione **norma** sono distinte dalle variabili locali a e b nella funzione **quad** (pur avendo lo stesso nome); infatti, quando per esempio la variabile a viene definita nelle funzione **quad**, essa è una variabile *locale* nella funzione **quad** e il compilatore la considera distinta dalla variabile a che compare nella funzione **norma**. (Se quest'ultima frase vi risulta ancora piuttosto oscura, non vi preoccupate, perché ritorneremo fra breve su questo concetto cercando di chiarirlo meglio con degli esempi.) Fino ad ora abbiamo considerato funzioni definite in C++ che sostanzialmente corrispondono al concetto di funzione come noi lo usiamo comunemente in matematica: si passano alla funzione i valori degli argomenti e in cambio la funzione fornisce un valore di ritorno. Tuttavia le funzioni in C++ sono qualcosa di più di questo (e di diverso); per esempio, è possibile definire funzioni che non restituiscono alcun valore. Ci possiamo chiedere: a cosa serve una funzione del C++ che non calcola alcun valore? Certo non può servire a calcolare, per esempio, nessuna funzione matematica, tuttavia essa potrebbe essere utilizzata

per eseguire qualche altro compito. Qui di seguito riportiamo una variante dell'Esempio 5.5 in cui viene usata la funzione **stampa**, il cui unico scopo è quello di stampare sul video il risultato finale del programma:

```
Esempio 5.6
//QUESTO PROGRAMMA CALCOLA LA RADICE QUADRATA DELLA SOMMA
//DEI QUADRATI DI DUE NUMERI X ED Y ASSEGNATI
 #include <math.h>
 #include <iostream>
 using namespace std;
  float norma(float,float);
  float quad(float);
  void stampa(float,float,float);
  main()
 {float x,y,z;
  cout << "\ndammi un numero reale x=";cin >> x;
  cout << "\ndammi un numero reale y=";cin >> y;
  z=norma(x,y);
  stampa(x,y,z);
  system("PAUSE");  return 0; }
  float norma(float a,float b)
 {float c;
  c=sqrt(quad(a)+quad(b));
  return c; }
  float quad(float a)
 {float b; b=a*a;
  return b; }
  void stampa(float a,float b,float c)
 {cout << "\n\nla norma di "<< a << " e " << b << " è " <<c;
  return; }
```

Osserviamo prima di tutto che, nel definire la funzione **stampa**, abbiamo dichiarato che il tipo della funzione è **void**; ciò significa che la funzione **stampa** non ritorna alcun valore e infatti nel corpo della funzione l'istruzione finale è semplicemente un **return;** senza alcun valore ad esso collegato. Vediamo ora come agisce la funzione **stampa** quando nel **main** si arriva all'istruzione **stampa(x,y,z);** vengono passati alla funzione **stampa** i valori delle variabili x, y e z, che vengono assegnati come valori iniziali delle variabili locali a, b e c rispettivamente della funzione; a questo punto la funzione **stampa** esegue la sua istruzione

```
cout << "\n\nla norma di " << a << " e " << b << " e' "<<c;
```

con cui viene stampato sul video il risultato del programma e successivamente l'istruzione **return;** termina l'esecuzione della funzione stampa.

Si osservi che, sulla base di quanto abbiamo già detto a proposito delle variabili locali, le variabili a, b e c della funzione stampa sono fisicamente distinte dalle variabili x, y e z del **main**; ciò implica che, se le istruzioni nella funzione stampa modificassero i valori di a, b e c, queste modifiche non avrebbero alcun effetto sui valori delle variabili x, y e z del **main**. Per comprendere meglio questi concetti, esaminiamo il seguente programma:

```
 Esempio 5.7
// QUESTO PROGRAMMA USA UNA FUNZIONE PER SCAMBIARE DUE
// NUMERI DATI; IL PASSAGGIO DEI PARAMETRI AVVIENE PER
// VALORE E LO SCAMBIO E' EFFETTUATO PERCIO' IN MODO SBAGLIATO
 #include <iostream>
 using namespace std;
  void scambia(int,int);
  main()
 {int x,y;
  cout << "\ndammi un numero reale x=";cin >> x;
  cout << "\ndammi un numero reale y=";cin >> y;
  cout << "\n\n  ecco i numeri prima dello scambio: ";
  cout << "\n\n    x=" << x << "   y=" <<y <<endl << endl;
  scambia(x,y);
  cout << "\n\n  ecco i numeri dopo lo scambio: ";
  cout << "\n\n    x=" << x << "   y=" <<y <<endl << endl;
  system("PAUSE");return 0; }
  void scambia(int a,int b)
 {int aux;
  aux=a;a=b;b=aux; return; }
```

(N.B.: l'istruzione **endl** è una istruzione equivalente a

```
\n
```

cioè quindi è una istruzione che semplicemente fa andare a capo.) In quest'ultimo programma viene utilizzata la funzione scambia (di tipo **void**); quando nel **main** si arriva alla istruzione **scambia(x,y);** vengono passati alla funzione i valori delle variabili x e y che vengono assegnati come valori alle variabili locali a e b; a questo punto la funzione **scambia** effettua lo scambio dei valori di a e b e questo scambio viene effettuato correttamente, cioè i

valori di a e di b si scambiano effettivamente, tuttavia tutto ciò non provoca alcun cambiamento sulle variabili x e y del **main**. Ed infatti, se compiliamo ed eseguiamo il programma, vediamo che i valori di x e y prima e dopo lo scambio sono gli stessi. Potremmo essere tentati di risolvere questo problema modificando il programma precedente al modo seguente:

```
Esempio 5.8
// QUESTO PROGRAMMA USA UNA FUNZIONE PER SCAMBIARE DUE
// NUMERI DATI; IL PASSAGGIO DEI PARAMETRI AVVIENE PER
// VALORE E LO SCAMBIO E' EFFETTUATO PERCIO' IN MODO SBAGLIATO
 #include <iostream>
 using namespace std;
  void scambia(int,int);
  main()
 {int x,y;
  cout << "\ndammi un numero reale x=";cin >> x;
  cout << "\ndammi un numero reale y=";cin >> y;
  cout << "\n\n  ecco i numeri prima dello scambio: ";
  cout << "\n\n    x=" << x << "   y=" <<y <<endl << endl;
  scambia(x,y);
  cout << "\n\n  ecco i numeri dopo lo scambio: ";
  cout << "\n\n    x=" << x << "   y=" <<y <<endl << endl;
  system("PAUSE");return 0; }
  void scambia(int x,int y)
 {int aux;
  aux=x;x=y;y=aux;
  return; }
```

Come si vede, la modifica è consistita nel dare lo stesso nome (x e y) sia alle variabili del **main** che a quelle della funzione **scambia**; tuttavia ciò non risolve affatto il problema, come è facile verificare compilando ed eseguendo anche quest'ultimo programma. La ragione di ciò sta nel fatto che, come abbiamo già avuto modo di dire, il fatto che le variabili x e y nel **main** e nella funzione **scambia** abbiano lo stesso nome, non significa affatto che esse (cioè le loro posizioni di memoria nel calcolatore) coincidano, e il compilatore le considera variabili distinte anche se hanno lo stesso nome; il risultato sarà pertanto che i valori delle variabili x e y nel **main** non si scambiano affatto, esattamente come accadeva nel programma dell' Esempio 5.7.

Tutti i nostri problemi nascono dal fatto che il passaggio dei valori delle variabili dal **main** alla funzione **scambia** avviene con la modalità detta *passaggio per valore delle variabili*, che è appunto il modo di passaggio che

abbiamo descritto in precedenza. In C++ esiste un altro modo di passare i valori delle variabili dal **main** ad una funzione, che è il cosiddetto *passaggio per indirizzo delle variabili*; senza entrare troppo nei dettagli (che verranno chiariti meglio nel seguito), possiamo dire che, quando si passa una variabile per indirizzo dal **main** ad una funzione, la variabile del **main** e la corrispondente variabile nella funzione sono fisicamente identiche, cioè occupano la stessa posizione nella memoria del calcolatore; ciò implica che ogni modifica alla variabile apportata nella funzione ha effetto anche sulla variabile del **main**. Il passaggio per indirizzo di una variabile si indica semplicemente aggiungendo il simbolo **&** al tipo della variabile, sia nel prototipo della funzione che nella testata della funzione. Ecco quindi come possiamo modificare il programma dell' Esempio 5.8 in modo che lo scambio avvenga in modo corretto:

```
 Esempio 5.9
// QUESTO PROGRAMMA USA UNA FUNZIONE PER SCAMBIARE DUE NUMERI
// DATI; IL PASSAGGIO DELLE VARIABILI AVVIENE PER INDIRIZZO
// E LO SCAMBIO E' EFFETTUATO PERCIO' IN MODO CORRETTO
 #include <iostream>
 using namespace std;
  void scambia(int&,int&);
  main()
 {int x,y;
  cout << "\ndammi un numero intero x=";cin >> x;
  cout << "\ndammi un numero intero y=";cin >> y;
  cout << "\n\n  ecco i numeri prima dello scambio: ";
  cout << "\n\n    x=" << x << "   y=" <<y <<endl << endl;
  scambia(x,y);
  cout << "\n\n  ecco i numeri dopo lo scambio: ";
  cout << "\n\n    x=" << x << "   y=" <<y <<endl << endl;
  return 0; }
  void scambia(int& a,int& b)
 {int aux;
  aux=a;a=b;b=aux;
  return; }
```

Una utilizzazione pratica del passaggio per indirizzo, potrebbe essere quella di servirsene per costruire funzioni che *leggono* valori, così come avevamo costruito una funzione che stampa valori (la funzione **stampa** nell' Esempio 5.6); ecco appunto come possiamo modificare in questo senso il programma dell' Esempio 5.6:

```
Esempio 5.10
 #include <math.h>
 #include <iostream>
 using namespace std;
 float norma(float,float);
 float quad(float);
 void stampa(float,float,float);
 void leggi(float&,float&);
 main()
{ float x,y,z;
  leggi(x,y);
  z=norma(x,y);
  stampa(x,y,z);
  system("PAUSE");return 0; }
  float norma(float a,float b)
{ float c;
  c=sqrt(quad(a)+quad(b));
  return c; }
  float quad(float a)
{ float b;
  b=a*a;
  return b; }
  void stampa(float a,float b,float c)
{ cout << "\n\nla norma di " << a << " e " << b << " è " <<c;
  return; }
  void leggi(float& a,float& b)
{ cout << "\ndammi un numero reale x=";cin >> a;
  cout << "\ndammi un numero reale y=";cin >> b;
  return; }
```

Come si vede, in questo caso abbiamo introdotto una funzione **leggi** (di tipo **void**, cioè quindi di un tipo che *apparentemente* non restituisce alcun valore) in cui le due variabili sono passate per indirizzo; in questo modo i valori che vengono letti (cioè assegnati tramite tastiera) per le variabili locali a e b nella funzione **leggi**, passano automaticamente nelle variabili x e y del **main**.

Come si vede dai precedenti esempi, c'è una profonda differenza fra il passaggio di variabili per valore e per indirizzo; tuttavia questa differenza si riferisce soltanto alle variabili di tipo **int**, **float**, **double**, **char** e alle loro varianti **short** e **long**.

Per quanto riguarda le variabili **array** (ad una o più dimensioni) invece, il C++ prevede solo il passaggio per indirizzo, nel senso che *comunque una variabile di tipo array viene sempre passata per indirizzo* in C++, anche se nel programma non viene usato l'operatore **&**.

Ecco un esempio di un programma che, sfruttando questa convenzione del C++, legge da tastiera un numero intero n e successivamente le componenti di un vettore di dimensione n che poi stampa sullo schermo:

```
Esempio 5.11
 #include <stdio.h>
 #include <stdlib.h>
 const int size=100;
 typedef double vettore[size];
 void letvet(vettore,int);
 void stampvet(vettore,int);
 main()
 {
  vettore v; int dim;
  printf("\nintroduci la dimensione del vettore: n=");
  scanf("%d",&dim);
  letvet(v,dim);
  stampvet(v,dim);
  system ("PAUSE");
  return 0;
 }
 void letvet(vettore vett,int n)
 { printf("introduci i dati\n\n");
   for (int i=1;i<=n;i++)
      {printf("\nv[%d]=",i);
       scanf("%lf",&vett[i]);}
   return; }
 void stampvet(vettore b,int n)
 { printf("\n\n\n   i dati introdotti sono i seguenti:\n\n");
   for (int i=1;i<=n;i++)
       printf("\n     v[%d]=%12.6f",i,b[i]);
   return; }
```

Per quanto riguarda il programma dell'Esempio 5.11 ci sono da osservare le seguenti cose:

1. l'istruzione

   ```
   const int size=100;
   ```

   serve a definire una costante; dal punto di vista pratico il nome di una costante identifica una porzione di memoria esattamente come il nome di una variabile e questa porzione di memoria ha una dimensione che dipende dal *tipo* della costante (in questo caso di tipo **int**, ma potrebbe essere di tipo **float** o **double** o **char**, ecc.) esattamente come accadeva per le variabili. La differenza fondamentale fra una variabile e una costante sta nel fatto (come peraltro suggerisce il nome stesso) che, mentre il contenuto di una variabile può essere modificato durante l'esecuzione del programma, il contenuto di una costante non può essere modificato durante l'esecuzione del programma.

   In generale una dichiarazione di costante avrà la seguente forma:

   const *tipo-della-costante nome-della-costante* = *valore-della-costante*;

2. l'istruzione

   ```
   typedef double vettore[size];
   ```

   è un'istruzione che definisce un nuovo tipo di dato (*tipo definito dall'utente*), che si chiama **vettore**. C'è da osservare che, in realtà, tramite il **typedef** noi non stiamo veramente *creando* un nuovo tipo di dato, ma piuttosto stiamo attribuendo un *nuovo nome* ad un tipo di dato che già esiste (in particolare in questo caso noi attribuiamo il nome **vettore** ad un **array** di variabili **double** di dimensione **size**).

   In generale una dichiarazione che definisce un nuovo tipo di dato avrà la seguente forma:

   typedef *tipo nome* [*eventuale-dimensione*];

   dove il *nome* è scelto da noi; per esempio, nel caso precedente avremmo potuto usare, invece del nome **vettore**, un qualsiasi altro nome: **vector**, oppure **Jeeves** o anche **Qfwfq**. E qui veniamo al punto della questione: l'uso del **typedef** ci permette di rendere *più leggibile* da chiunque il nostro programma, infatti chiunque lo legga capirà subito che stiamo memorizzando dei vettori (in questo senso sarebbe stata quindi opportuna anche la scelta del nome **vector**, ma non quella di **Jeeves** o di **Qfwfq**).

3. Sia nel prototipo che nel corpo della funzione **leggi**, la variabile di tipo **vettore** viene passata senza l'uso del simbolo **&**, tuttavia il passaggio avviene comunque per indirizzo perchè le variabili di tipo **vettore**, come risulta dall'istruzione **typedef**, sono degli array.

Per concludere questa breve introduzione alle funzioni del C++, possiamo aggiungere che è possibile anche definire funzioni che non hanno alcun argomento. Ecco un possibile uso di una funzione di questo tipo:

Esempio 5.12

```
 #include <math.h>
 #include <iostream>
 using namespace std;
 float norma(float,float);
 float quad(float);
 void stampa(float,float,float);
 void leggi(float&,float&);
 void presenta();
  main()
 {float x,y,z;
  presenta();system("PAUSE");
  leggi(x,y); z=norma(x,y);
  stampa(x,y,z);system("PAUSE"); return 0; }
 float norma(float a,float b)
  {float c;c=sqrt(quad(a)+quad(b));return c; }
 float quad(float a)
  {float b; b=a*a;return b; }
 void stampa(float a,float b,float c)
  {cout << "\n\nla norma di "<< a << " e " << b << " è " <<c;
   return; }
 void leggi(float& a,float& b)
  {cout << "\ndammi un numero reale x=";cin >> a;
   cout << "\ndammi un numero reale y=";cin >> b;
   return; }
 void presenta()
  {cout << "\n  QUESTO PROGRAMMA CALCOLA LA RADICE QUADRATA";
   cout << "\n  DELLA SOMMA DEI QUADRATI DI DUE NUMERI X ED Y";
   cout << "\n  ASSEGNATI. SE SEI PRONTO A FORNIRMI I VALORI";
   cout << "\n  DI X E Y PREMI UN TASTO QUALSIASI";
   return; }
```

Come si vede la funzione **presenta** (di tipo **void**, cioè che non restituisce alcun valore) ha esclusivamente lo scopo di mandare inizialmente sullo schermo video un messaggio che presenta all'utilizzatore lo scopo del programma e quello che ci si aspetta da lui; lo schermo si immobilizza poi in questa posizione per effetto della istruzione

```
system("PAUSE");
```

fino a che l'utilizzatore non preme un tasto qualsiasi in modo che il programma possa riprendere la sua esecuzione.

A questo punto qualcuno potrebbe chiedersi: forse

```
system("PAUSE");
```

(che è definita nel file **stdlib.h**) è anch'essa una funzione che ha come argomento **PAUSE**? La risposta è: si, è proprio così. E infine, qualcuno potrebbe anche chiedersi: ma forse anche **main()** potrebbe essere una funzione senza alcun argomento (come **presenta**)? E la risposta è ancora: si, è proprio così; **main()** è la testata della funzione e il corpo della funzione è tutto ciò che è racchiuso fra le parentesi graffe aperta e chiusa che seguono. Qualcuno potrebbe obiettare: ma quale è il tipo della funzione **main()**? (Come sappiamo infatti, nella testata di una funzione dovrebbe comparire il nome della funzione preceduto dal suo tipo). Naturalmente il tipo della funzione **main()** non può essere **void**, infatti la funzione **main()** termina con l'istruzione **return 0;** che restituisce il valore 0. E infatti la funzione **main()** è di tipo **int**, soltanto che il tipo può, in questo caso particolare, essere omesso; tuttavia, se provate a modificare tutti i programmi che abbiamo scritto fino ad adesso sostituendo **main()** con **int main()** vedrete che tutto funzionerà perfettamente allo stesso modo.

Per concludere quindi, ogni programma C++ è costituito dalla funzione **main()** ed (eventualmente) da altre funzioni che vengono chiamate all'interno della funzione **main()** e l'esecuzione del programma ha sempre inizio con una chiamata della funzione **main()**.

Appendice A

Come aprire un file e scriverci dentro

Ecco un breve programma C++ che serve per aprire un file e scriverci dentro qualcosa:

```
Programma 1
 #include <stdio.h>
 #include <stdlib.h>
  int f(int);
  main()
  { FILE *fp;
    fp=fopen("ecco.dat","w+");
    if (fp==NULL) { printf("impossibile");
                    system("PAUSE");exit(1);}
    int x,y;x=8;y=f(x);
    fprintf(fp,"\n   x=%d    y=%d",x,y);
    fclose(fp);
    return 0; }
    int f(int k)
    {k=k-1;int w;
     w=k+k;return w;}
```

in particolare il gruppo di istruzioni:

```
  FILE *fp;
  fp=fopen("ecco.dat","w+");
  if (fp==NULL) { printf("impossibile");
                    system("PAUSE");exit(1);}
```

serve per creare il file di nome **ecco.dat**, mentre l'istruzione

```
fprintf(fp,"\n    x=%d     y=%d",x,y);
```

serve a scrivere nel file **ecco.dat**; la funzione fprintf è essenzialmente identica alla funzione printf, con l'unica differenza che, invece di stampare sullo schermo, stampa nel file corrispondente ad **fp**, cioè, in questo caso, nel file **ecco.dat**. Dopo avere completato le operazioni di stampa su file, è sempre opportuno aggiungere l'istruzione

```
fclose(fp);
```

Se proviamo a compilare ed eseguire il programma 1, apparentemente sembrerà che non sia successo niente, tuttavia qualcosa è successo, anche se noi non lo vediamo immediatamente, appunto perchè l'uscita dei dati in questo caso è avvenuta sul file **ecco.dat** e non sullo schermo (osservate in particolare che in questo caso non abbiamo posto il solito

```
system("PAUSE");
```

prima dell'istruzione finale **return 0**); per vedere i risultati del nostro programma dobbiamo quindi andare a cercare il file **ecco.dat** e aprirlo per vedere cosa c'è dentro.

Se voi state lavorando (come è consigliabile) nella directory Home, troverete facilmente il file **ecco.dat** in questa directory.

Provate ora a sostituire l'istruzione

```
fp=fopen("ecco.dat","w+");
```

con l'istruzione

```
fp=fopen("ecco.dat","a");
```

e ricompilate ed eseguite più volte il programma; che cosa cambia rispetto al caso precedente?

Studiate ora il seguente programma e cercate di capire che cosa fa.

```
Programma 2
 #include <stdio.h>
 #include <stdlib.h>
 #include <math.h>
  double f(double t)
  { return sin(t)*cos(t); }
  main()
  { double x,y; double a=1,b=6;
    int n=200; double h=(b-a)/n;
    FILE *fp;
    fp=fopen("tabella.dat","w+");
    if (fp==NULL) { printf("impossibile");
                    system("PAUSE");exit(1);}
    for (int i=0;i<=n;i++)
       { x=a+i*h;
         y=f(x);
         fprintf(fp,"  %f    %f  \n",x,y); }
    fclose(fp);
    return 0;  }
```

Provate a compilarlo ed eseguirlo.

Come avrete constatato, il Programma 2 crea un file di nome **tabella.dat**, e ci scrive dentro due colonne di numeri: la prima colonna è costituita dai valori

$$x_i \quad per \quad i = 0, 1, \ldots, 200$$

ottenuti suddividendo l'intervallo $[1, 6]$ in 200 parti uguali e la seconda colonna è costituita dai corrispondenti valori $f(x_i)$, cioè, in questo caso , dai valori $sinx_i \cdot cosx_i$.

Come esercizio, provate a modificare il precedente programma in modo da ottenere un programma che fa le seguenti cose:

1. legge (da tastiera) due numeri reali a e b
2. legge (da tastiera) un numero intero n
3. suddivide l'intervallo $[a, b]$ in n parti uguali e stampa nel file **risultati.dat** una tabella di due colonne, in cui la prima colonna è costituita dai punti di suddivisione x_i per $i = 0, 1, \ldots, n$ e la seconda colonna è costituita dai valori corrispondenti della funzione

$$\frac{cosx}{1 + sin^2 x}$$

E' possibile aprire ed utilizzare in uno stesso programma due o più file in cui andare a scrivere qualcosa; naturalmente, per fare ciò dovremo utilizzare nomi diversi per ciascun file e indicatori diversi per ciascun file.

Ecco qui di seguito un esempio ottenuto modificando il precedente Programma 2 in modo da andare a scrivere in differenti file i valori delle ascisse e i valori delle ordinate della funzione f(x):

```
Programma 2
 #include <stdio.h>
 #include <stdlib.h>
 #include <math.h>
  double f(double t)
  { return sin(t)*cos(t); }
  main()
  { double x,y; double a=1,b=6;
    int n=200; double h=(b-a)/n;
    FILE *fp1,*fp2;
    fp1=fopen("x.dat","w+");
    if (fp1==NULL) { printf("impossibile");
                     system("PAUSE");exit(1);}
    fp2=fopen("y.dat","w+");
    if (fp2==NULL) { printf("impossibile");
                     system("PAUSE");exit(1);}
    for (int i=0;i<=n;i++)
       { x=a+i*h;
         y=f(x);
         fprintf(fp1,"  %f  \n",x); }
         fprintf(fp2,"  %f  \n",y); }
    fclose(fp1);
    fclose(fp2);
    return 0;  }
```

Appendice B

Usare GNUplot

Quando avviate GNUplot, la prima cosa che di solito conviene fare è quella di cambiare la directory di default con la directory in cui si trovano i file che volete plottare. Per fare questo basta cliccare sul pulsante ChDir (che si trova sulla maschera principale di GNUplot) e successivamente digitare la directory che volete (per esempio, se lavorate con uno dei PC del Laboratorio di Calcolo, tipicamente dovrete dare il percorso che porta alla cartella Home, che è la cartella dove di solito andranno a finire i file che create attraverso un programma C++).

A questo punto potete provare a plottare un file; l'istruzione basilare per fare ciò è la seguente:

```
plot 'nomefile'  with lines
```

dove *nomefile* deve essere il nome di un file che si trova nella cartella della directory che avete scelto (attenzione: *nomefile* deve essere racchiuso fra apici) e inoltre questo file deve contenere due colonne di numeri separate da almeno uno spazio bianco e niente altro.

Se avete fatto tutto correttamente, a questo punto, schiacciando il pulsante return vedrete comparire il grafico di una funzione che è ottenuta interpretando i valori della prima colonna del file *nomefile* come valori sull'asse x e i valori della seconda colonna come i corrispondenti valori sull'asse y.

Se il file *nomefile* contiene una tabella di numeri costituita da più di due colonne, è possibile specificare quale delle colonne va interpretata come asse x e quale come asse y; per esempio, supponiamo che il nostro file contenga quattro colonne di numeri (separate da almeno uno spazio bianco), in questo caso possiamo disegnare il grafico della funzione ottenuta interpretando la seconda colonna come asse x e la quarta come asse y con la seguente istruzione GNUplot:

```
plot 'nomefile' using 2:4 with lines
```

E' anche possible plottare contemporaneamente due funzioni; per esempio se il file nomefile contiene una tabella di numeri costituita da 3 colonne, usando l'istruzione

```
plot 'nomefile' using 1:2 with lines,'nomefile' using 1:3 with
lines
```

si ottengono contemporaneamente i grafici delle funzioni che interpretano la prima colonna come asse x e rispettivamente la seconda e terza colonna coma asse y; di default i due grafici saranno disegnati con colori diversi (se si sceglie di usare solo il bianco e nero saranno disegnati usando una linea continua ed una linea tratteggiata). Naturalmente è possibile, con lo stesso principio, disegnare sullo stesso grafico tre o più funzioni.

Esercizio proposto: Provate a disegnare il grafico delle seguenti funzioni

1. $\sin x \cdot \cos x$
2. $\frac{\cos x}{1+\sin^2 x}$
3. $\arctan \frac{1}{(1+x^2)}$

nell'intervallo $[-4, 4]$.