Fondamenti di C++

Obiettivi del tutorial.

In questa guida apprenderete le basi della programmazione in C++; partiremo da zero ed arriveremo a capire come usare funzioni, array e matrici.

Prima di iniziare…

Dato che partiremo dalle basi non sono richieste conoscenze preliminari, mentre sul fronte software vi dovrete procurare un "compilatore", ossia un "programmino" che interpreta il codice che scriverete e darà vita alle vostre applicazioni, se non ricordo male è disponibile su www.borland.com alla pagina download.

Primi passi…

Lo strumento di cui ci dobbiamo servire ancora prima di scrivere un programma è "l'algoritmo". Definiamo algoritmo una sequenza di passi elementari non ambigui che operando su una serie di dati iniziali restituisce un risultato. E' importante saper creare buoni algoritmi perché scrivere un programma significa sostanzialmente risolvere dei "problemi", quindi trovare una soluzione logica ai problemi è la chiave per programmare correttamente. Facciamo un esempio di algoritmo: se voglio fare un programma che calcola l'area di un quadrato l'algoritmo corrispondente è il seguente:

1- chiedere all'utente di inserire la misura del lato e memorizzarlo in memoria
2- moltiplicare il lato per se stesso e memorizzare il risultato
3- restituire su schermo il risultato.

Quindi possiamo notare come siano dei passi semplicissimi che non possono essere fraintesi. Un altro aspetto da tenere in considerazione è che questa sequenza non deve essere infinita, ma deve giungere ad una conclusione.
Vediamo un altro esempio di algoritmo, questa volte leggermente più complesso. Voglio un programma che calcola il maggiore fra 2 numeri:

1- chiedi all'utente di inserire un numero e memorizzalo;
2- chiedi all'utente un secondo numero e memorizzalo;
3- calcola la differenza fra il primo ed il secondo numero e memorizzala;
4- se la differenza è maggiore di zero esegui 5, altrimenti esegui 6;
5- mostra su schermo: "Il numero massimo è…" seguita dal primo numero inserito, vai a 7;
6- mostra su schermo: "Il numero massimo è…" seguita dal secondo numero inserito;
7- fermati.

Al passo 4 c'è un particolare tipo di istruzione detta "condizionale", che permette di eseguire una istruzione solo se sono verificate determinate condizioni.

Ora facciamo un algoritmo simile al precedente, ma questa volta con tre numeri.

1- chiedi il primo numero e chiamalo X;
2- chiedi il secondo numero e chiamalo Y;
3- chiedi il terzo numero e chiamalo Z;
4- se X è maggiore di Y (per farlo si usa l'algoritmo di prima) vai a 5, altrimenti vai a 6;
5- calcola il massimo tra X e Z e mostra su schermo il risultato. Vai a 7;
6- calcola il massimo tra Y e Z e mostra il risultato su schermo. Vai a 7;
7- fermati.

Possiamo notare due particolarità: primo, ho usato ancora una volta una istruzione condizionale, secondo, ho fatto riferimento ad un altro algoritmo, questo non è stato fatto a caso, ma è una caratteristica della programmazione moderna, infatti se dobbiamo risolvere un problema complesso lo dividiamo in tanti piccoli problemi e ne cerchiamo la soluzione.
Se infine il massimo non lo voglio tra 2, 3 o 4 numeri, ma tra n numeri? Allora confronto il primo col secondo, il massimo fra questi lo confronto col terzo e così via per n volte. Questa istruzione è detta "ciclica"(o iterativa), e la approfondiremo più avanti.
A questo punto è chiaro che per scrivere un programma serve un efficace algoritmo, quindi prima di mettere mano al codice un problema deve essere risolto in maniera semplicemente logica, poi bisogna creare un algoritmo tenendo conto di ciò che sa fare il computer e solo alla fine mettere mano al codice.

Il nostro primo programma in C++

Di seguito è riportato un programma che mostra su schermo "Welcome to C++"; verrà illustrato alla fine riga per riga.

```
// Primo programma in C++
#include <iostream.h>
int main()
{
        cout<<"Welcome to C++! \n";
        return 0; // indica che il programma è terminato con successo
}
```

Scrivendo questo codice nel compilatore si avrà un programma che mostra "Welcome to C++". Analizziamolo riga per riga.

1- La prima riga inizia con //: questo indica che c'è un commento; i commenti servono ai programmatori per spiegare a cosa serve il codice scritto, dato che serve solo al programmatore esso non sarà interpretato dal compilatore, che semplicemente lo ignorerà. Possono essere messi dove si vuole l'importante è ricordare che è considerato un commento tutto ciò che segue le // e sta sulla stessa riga; se volessimo scriverlo su più righe allora dovremmo scrivere:
```
// Primo
// programma
// in C++
```
2- Come si può intuire il comando #include serve per "includere" un qualcosa che è specificato tra < >; tra le parentesi ci sono inseriti i nomi di "librerie", dato che sull'argomento ci torneremo più avanti per ora sappiate solo che "iostream.h" è indispensabile per gestire le informazioni in entrata ed in uscita.
3- In un certo senso "int main" significa che inizia il programma, anche per questo comando tutto sarà più chiaro quando andremo piuttosto avanti.
4- Da qui in poi inizia il codice vero e proprio.
5- Il comando "cout" sta per "standard output" ed in particolare ci serve per mostrare su schermo ciò che è compreso fra gli apici, i << immaginateli come frecce che portano sullo schermo(il cout) quello che c'è a destra. Il comando \n indica di andare a capo, ed è consigliabile inserirlo per una questione di chiarezza, notate che la \ è un comando di "escape" ed indica che ciò che c'è dopo è da interpretare(e non lo confondete con /…). Oltre a \n ci sono altri tipi di comandi, in particolare: \t fa una "tabulazione" ossia lascia un po' di

spazio sulla stessa riga, \r riporta il cursore a inizio riga, \a fa un suono alla fine della scrittura; se voglio mostrare su schermo una \ ne metto 2 (così: \\) se volessi mostrare degli apici metto \", tutte queste istruzioni possono essere combinate e scrivere per esempio \a\n. Alla fine c'è un punto e virgola, che semplicemente va messo alla fine di ogni istruzione.

6- return 0 indica che il programma è terminato e deve restituire qualcosa, esso è collegato ad int main e quindi il suo ruolo vi sarà chiaro più avanti. Dopo return 0 ho messo un commento, e vedo che non influisce essendo dopo le istruzioni; attenti a non scrivere mai cose del tipo: *//return 0; indica che il programma è terminato*
poiché è un errore gravissimo, dato che il compilatore non interpreta ciò che sta dopo le //.

Se al posto di

cout<<"Welcome to C++\n";

avessimo scritto:

cout<<"Welcome ";
cout<<"to C++ \n";

non ci sarebbe stata alcuna variazione nel risultato finale, dato che il compilatore avrebbe prima scritto welcome e poi to C++ sulla stessa riga.
Se avessimo scritto:

cout<<"Welcome \n to \n \n C++ \n";

allora avremmo avuto questa scritta:

Welcome
to

C++

Le variabili

Per spiegare che cosa è una *variabile* procedo come in precedenza: scrivo un programma e ne spiego ogni singola riga.

```
//Programma che calcola la somma di 2 interi
#include <iostream.h>
int main()
{
        int intero1, intero2, somma; //dichiarazione

        cout<<"Inserisci il primo intero \n";
        cin>>intero1;

        cout<<"Inserisci il secondo intero \n";
        cin>>intero2;

        somma = intero1 + intero2;
        cout<<"La somma è "<<somma<<endl;
        return 0;
```

}

Questo programma chiede all'utente di inserire due numeri interi, ne calcola la somma, e quindi mostra il risultato su schermo. Analizziamolo riga per riga.

1- E' un commento, e lo abbiamo già visto in precedenza;
2- Includiamo la libreria che ci serve(a noi servirà sempre la iostream.h);
3- Inizia il programma;
4- Una semplice parentesi…
5- Come leggete nel commento questa è una "dichiarazione", per spiegare il suo ruolo bisogna fare un discorso più ampio del solito. Quando dobbiamo scrivere un programma noi dobbiamo lavorare su dei dati, quindi abbiamo bisogno di occupare un certo spazio nella memoria del nostro PC, e in questa memoria salveremo le informazioni che ci servono. Lo spazio che chiediamo alla CPU lo usiamo per memorizzare delle *variabili*(che nel nostro caso abbiamo chiamato intero1, intero2, somma), quindi la *variabile* è una specie di "celletta" in cui memorizzare un dato. Ci sono diversi tipi di *variabili* e ad ogni tipo di *variabile* corrisponde un tipo di dato, quello che usiamo qui è un intero, quindi nelle cellette intero1, intero2 e somma possono essere memorizzati solo numeri interi. Altri tipi li vedremo in seguito. Per dichiarare una *variabile* si usa la parola *int*, che indica il tipo di *variabile* che ci serve, seguita dal nome che vogliamo dare a queste *variabili*; possiamo usare quante *variabili* vogliamo l'importante è che siano separate dalla virgola e da uno spazio, che non inizino con un numero, inoltre i nomi composti devono essere uniti da qualche carattere(per esempio posso dichiarare la *variabile_uno*). Come di consueto alla fine ci vuole un punto e virgola.
6- E' uno spazio vuoto, lo si può usare per rendere più leggibile un programma.
7- Questo comando già lo conosciamo.
8- Questo è un comando nuovo: *cin* serve per fare inserire un dato all'utente tramite la tastiera, le >> immaginatele come frecce che dall'utente si dirigono alla *variabile*, in cui salviamo il dato; infine c'è il nome della variabile in cui memorizzare il dato. Quindi il consueto punto e virgola.
9- Ancora vuoto
10- Ancora cout
11- Ancora cin
12- Ancora vuoto
13- Ora c'è una istruzione, che permette di memorizzare nella *variabile* chiamata "somma" la somma di intero1 e intero2; è molto intuitiva essendo la stessa scrittura che si usa in aritmetica. L'operatore = è noto anche come operatore di assegnamento, poiché assegna alla *variabile* a sinistra, il contenuto di ciò che c'è a destra; è ovvio che qualunque dato memorizzato precedentemente con questa operazione viene soprascritto. Notate come la CPU, per quanto stupida, sa eseguire le operazioni elementari, quindi addizioni, sottrazioni, moltiplicazioni e divisioni. Alla fine, come sempre, ci va il punto e virgola.
14- Infine ecco il cout, ora però ci sono delle note da aggiungere. Dopo "la somma è " c'è la variabile somma dopo <<, ma senza gli apici, il motivo è semplice: se noi mettiamo gli apici viene mostrato su schermo ciò che abbiamo scritto, quindi se scriviamo:

cout<<"somma";

su schermo si legge la parola somma, ma se non ci sono gli apici la CPU traduce ciò che abbiamo scritto, quindi se scriviamo:

cout<<somma;

su schermo verrà mostrato il contenuto di somma, qualunque esso sia (può essere -201 come 2001453, non c'è nessuna differenza). Dato che la CPU interpreta ciò che è scritto senza apici noi avremmo potuto scrivere anche:

cout<<"La somma è "<<intero1 + intero2<<endl;

le scritture portano alla stessa conclusione. Infine c'è da notare <<endl; questa è una alternativa a \n, quindi semplicemente porta il cursore a capo.

15- Return 0 lo conosciamo.
16- La parentesi indica che è finito il programma.

L'ultimo dettaglio da notare è che il compilatore esegue i comandi alla lettera ed in perfetto ordine, cioè parte dalla prima riga, poi va alla seconda e fa tutti i singoli passi sino ad arrivare alla fine, è ovvio che se ci sono errori, non sempre il compilatore lo segnala.

Mettetevi alla prova

Questa è una parte fondamentale del tutorial, nella quale vengono proposti degli esercizi con cui mettervi alla prova in prima persona. Vi consiglio di non andare avanti se non riuscite a risolvere questi esercizi.

1- Scrivere un programma C++ che legge(che chiede all'utente) i tre lati di un triangolo scaleno e restituisce(mostra su video) il perimetro.
2- Scrivere un programma, che chiede all'utente di immettere le 2 basi e il lato obliquo di un trapezio isoscele e, utilizzando semplicemente l'operatore +, restituisce il perimetro del trapezio.
3- Scrivere un programma che legge il lato di un quadrato e mostra su schermo il suo perimetro e l'area.

Notate che si può sommare il contenuto di una variabile per se stessa(per esempio posso scrivere A = B + B) oppure fare la somma di più di 2 interi(per esempio A = B + C + D + B).
Se userete un compilatore potrete verificare la correttezza degli esercizi di persona, ecco perché è consigliato.

Le costanti

Le "costanti" sono dette "variabili a sola lettura", infatti il loro valore non può essere in alcun modo modificato. Vediamo un programma che ne fa un corretto utilizzo.

```
//Utilizzo corretto di una costante
#include <iostream.h>
int main()
{
        const int x=7;
        cout<<"Il valore della costante è "<<x<<endl;
        return 0;
}
```

Dobbiamo notare come per dichiarare una costante basta semplicemente scrivere "const" prima del tipo di variabile. Ancora notiamo come alla costante abbiamo attribuito subito un valore: questo valore non può essere cambiato e se tentiamo di cambiarlo il compilatore segnala un errore.

```
//Utilizzo errato di una costante
#include <iostream.h>
int main()
{
        const int x;
```

```
        x=7;
        cout<<"Il valore della costante è "<<x<<endl;
        return 0;
}
```

Questo è il metodo con cui NON possono essere usate le costanti, infatti il loro valore deve essere dichiarato come più sopra.
Per dichiarare più costanti possiamo fare:

const int x=4, y=5;

oppure

const int x=4;
const int y=5;

Operazioni

Le operazioni fondamentali che è possibile sfruttare sono: addizione, sottrazione, moltiplicazione, divisione e modulo.

Addizione,sottrazione, moltiplicazione: non credo ci sia niente da spiegare; si indicano rispettivamente con +, -, *
Divisione: si indica con /. Dobbiamo notare che se memorizziamo il risultato di una divisione con resto diverso da zero in una variabile *int* ne conserveremo solo la parte intera. Esempio: 5/2 = 2.5, ma eseguendo *int a = 5/2* allora *a* sarà uguale a 2.
Modulo: si indica con %, questa operazione restituisce il resto di una divisione; per esempio 17%5=2.

Tramite le parentesi tonde è possibile creare delle espressioni che seguono le comuni convenzioni matematiche, per esempio:
((a + (b – 3) * 4) + 5 * (2 – 4))
le classiche parentesi quadre e graffe sono sostituite da altre tonde per il resto è tutto standard, quindi la CPU da priorità alle moltiplicazioni, divisione e modulo, e poi esegue le addizioni/sottrazioni, ed esegue prima le operazioni tra parentesi.

Le potenze per ora le eseguiamo come serie di prodotti ($x^3 = x * x * x$)

In C++ alcune scritture possono essere semplificate, per esempio:

il comando *c=c+5*, può essere compattato in *c+ =5* ed è come dire che incremento c di 5. Lo stesso discorso si può fare per esempio con:

c=c+d; ➔ *c+=d;*

questo vale anche per le altre operazioni elementari.

Incremento e decremento.

Di particolare importanza è l'incremento o decremento di una variabile di una unità. Per scrivere *C+=1* scriviamo direttamente *c++*(da cui il nome del linguaggio) ed al posto di *c-=1* si ha *c--*.
Esistono diversi tipi di incremento: il pre-incremento e il post-incremento. Il pre-incremento si indica con "++c": con questo comando prima si incrementa la variabile e poi si usa, il post-

incremento (c++) è il viceversa, cioè prima si usa la variabile e poi la si incrementa. Facciamo un esempio per chiarire la differenza, ed attenzione ai commenti:

```
//Esempi di pre-incremento e post-incremento
#include <iostream.h>
int main()
{
        int c;
        c=5;
        cout<<c<<endl; //con questa istruzione sullo schermo si vede 5
        cout<<c++<<endl; //sullo schermo riapparirà 5, e dopo c diventa 6
        cout<<c<<endl; //ora visualizza 6
        c=5;
        cout<<c<<endl; // si vede 5
        cout<<++c<<endl; //ora prima la c si incrementa a 6 e dopo viene mostrato su video
        //quindi si vede un bel 6
        cout<<c<<endl; //vedo ancora un bel 6
        return 0;
}
```

Operatori relazionali

Gli operatori relazionali ci permettono di verificare se delle condizioni sono verificate. Il loro utilizzo lo vedremo nel prossimo paragrafo, quindi per ora sappiate che:
minore si indica con <
maggiore con >
minore o uguale con <=
maggiore o uguale con >=
uguale con = = (da non confondere con =)
diverso con !=

Struttura selettiva: IF

L'istruzione if (che in inglese vuol dire "se") è un "istruzione di selezione", che permette di eseguire una operazione solo se è verificata una condizione; è strutturata in questo modo.

```
If(condizione)
{
        istruzione
}
```

Vediamo un esempio:

```
//Programma che legge un intero e determina se è maggiore di 10
#include <iostream.h>
int main ()
{
        int a;
        cout<<"Inserisci un intero"<<endl;
        cin>>a;

        if(a>10)
        {
```

```
            cout<<"Il valore inserito è maggiore di 10"<<endl;
        }

        return 0;
}
```

Seguiamo in ogni passo ciò che legge il compilatore.

1. Libreria, dichiarazioni, nulla di nuovo
2. Abbiamo dichiarato la variabile *a*, ed abbiamo chiesto all'utente di dargli un valore.
3. Supponiamo l'utente abbia inserito 20, allora il compilatore leggendo "if" calcola: *a*(che per noi è 20) è maggiore di 10? Si, allora eseguo ciò che c'è dentro l'if. Verrà mostrato su schermo il cout. Notate come l'istruzione da eseguire con l'if è racchiusa fra parentesi graffe.
4. Supponiamo ora che l'utente abbia inserito 5. Il compilatore dice: *a* è maggiore di 10? No, allora salto quello che c'è dentro l'if. Quindi su schermo non viene mostrato niente.

All'interno di un *if* possono essere inseriti altri *if*, per verificare per esempio più condizioni.
Ora vediamo un altro esempio.

```
//Programma che ordina due numeri in modo crescente
#include <iostream.h>
int main()
{
        int num1, num2, t; //la variabile t la usiamo come supporto
        cout<<"Inserisci 2 interi"<<endl;
        cin>>num1>>num2; //invece di scrivere due cin ne abbiamo usato uno solo

        if(num1>num2)
        {
                t=num1;
                num1=num2;
                num2=t;
        }

        //nell'if ho effettuato lo scambio tra due numeri. Per capire con un esempio semplice ciò che
        //ho fatto supponiamo di avere due bicchieri (che nel codice sono num1 e num2) uno con
        //acqua e l'altro con birra, per scambiarne il contenuto usiamo un terzo bicchiere, quindi
        //travasiamo l'acqua nel terzo bicchiere, poi mettiamo la birra nel bicchiere che era pieno
        //d'acqua, e infine mettiamo l'acqua dal terzo bicchiere a quello che era pieno di birra

        cout<<num1<<" "<<num2<<endl; //ora vengono mostrati i numeri in ordine, da notare
        //questa scrittura che ci risparmia due righe di cout.
        return 0;
}
```

Facciamo ancora un altro esempio.

```
//Programma che calcola il massimo tra due numeri
#include <iostream.h>
int main()
{
        int a, b, max;
```

```
        cout<<"Inserisci due interi"<<endl;
        cin>>a>>b;

        max=a;

        if(max<b)
                max=b;

        cout<<"Il massimo è "<<max<<endl;

        return 0;
}
```

In questo programma poniamo il primo numero come massimo a priori e poi lo confrontiamo col secondo intero, SE l'intero fosse stato più grande del massimo allora sarebbe diventato il mio massimo. Si noti come l'istruzione nell'*if* non è racchiusa fra parentesi graffe: se nell'if c'è una sola riga di codice le parentesi si possono evitare.

Mettetevi alla prova

Eseguite questi piccoli esercizi e poi andate avanti:

1- Per un programmatore è importante saper interpretare il codice che ha davanti a se, pertanto in questo primo esercizio dovete capire che cosa fa il programma:

```
#include<iostream.h>
int main()
{
        int a;
        cout<<"Inserisci un intero"<<endl;
        cin>>a;

        if(a>5)
        {
                cout<<a<<endl;
                a++;
        }

        if(a==10)
                cout<<"Bingo!!"<<endl;

        return 0;
}
```

2- Scrivere un programma che legge un intero e stabilisce se è maggiore, minore o uguale a zero
3- Scrivere un programma che legge tre interi e ne determina il massimo
4- Scrivere un programma che legge tre interi e determina il minore.

If – else

Quando abbiamo usato l'if abbiamo fatto in modo che quando una condizione risulta vera vengono eseguite delle operazioni, altrimenti queste vengono saltate. Attraverso l'istruzione *else*("altrimenti")

possiamo fare in modo che se non è verificata una condizione nell'if, allora vengano eseguite delle operazioni. Vediamo un esempio:

```
//Programma che stabilisce se uno studente è promosso o bocciato
#include<iostream.h>
int main()
{
	int voto;
	cout<<"Inserisci il voto dello studente"<<endl;
	cin>>voto;

	if(voto>=18)
		cout<<"Promosso"<<endl;
	else
		cout<<"Bocciato"<<endl;
	return 0;
}
```

Quindi SE il voto è >= 18 si legge promosso, ALTRIMENTI si legge bocciato. Si noti come viene eseguita solo una delle istruzioni, infatti se la condizione (voto>=18) è verifica viene eseguito il *cout<<"Promosso"<<endl;* e si va a return 0.
Molte volte le condizioni che possono verificarsi possono essere più di 2, in tal caso possiamo inserire altri if dopo l'else. Esempio:

```
//Programma che dato un voto da 0 a 30, lo converte nella votazione americana, con:
// A = 30 a 27, B = 26 a 24, C = 23 a 21, D = 20 a 18, F = 17 a 0.
#include<iostream.h>
int main()
{
	int voto;
	cout<<"Inserisci il voto"<<endl;
	cin>>voto;

	if(voto>=27)
		cout<<"A"<<endl;
	else if(voto>=24)
		cout<<"B"<<endl;
	else if(voto>=21)
		cout<<"C"<<endl;
	else if(voto>=18)
		cout<<"D"<<endl;
	else
		cout<<"F"<<endl;

	return 0;
}
```

In questo caso sono "annidati" vari *if-else*, in modo da avere un valore per ogni caso. Si noti come la CPU nell'eseguire le istruzioni faccia un passo per volta leggendo le istruzioni nell'ordine in cui sono scritte.

Mettetevi alla prova

1- Che cosa fa questo programma?

```
#include<iostream.h>
int main()
{
        int a;
        cout<<"Inserisci un intero"<<endl;
        cin>>a;

        if(a>100)
                cout<<"Wow"<<endl;
        else if(a>50)
                cout<<"Mica male…"<<endl;
        else
                cout<<"Scarsetto"<<endl;

        a*=10;

        if(a<500)
                cout<<"Fa, niente…"<<endl;

        return 0;
}
```

2- Scrivere un programma che legge un intero e determina se è pari o dispari. Suggerimento: usare l'operatore modulo, in particolare un numero pari è sempre multiplo di 2, e i multipli di 2 danno resto 0 se sono divisi per 2.
3- Scrivere un programma che legge 2 interi e determina se il primo intero è multiplo del secondo. Suggerimento: utilizzare l'operatore modulo.

Operatori logici

Fino ad ora nelle istruzioni abbiamo imposto un'unica condizione. Tuttavia si rende necessario, a volte, verificare che siano soddisfatte 2 o più condizioni. Per fare ciò si usano gli operatori logici. Ogni operatore può restituire true(vero) oppure false(false). Vediamo quali sono e come si usano.

&&: è l'operatore "and", che significa "e". Serve per imporre che siano verificate contemporaneamente più condizioni. Se per esempio voglio fare un programma che verifica se un intero è contemporaneamente multiplo di 2 e 3, faccio:

```
#include<iostream.h>
int main()
{
        int a;
        cout<<"Inserisci intero"<<endl;
        cin>>a;

        if(a%2==0 && a%3==0)
                cout<<"L'intero è un multiplo"<<endl;
        else
                cout<<"L'intero non è un multiplo"<<endl;
```

```
	return 0;
}
```

Si noti come viene eseguito l'*if* solo se sono verificate ENTRAMBE le condizioni, basta che solo una non sia verificata e non viene eseguito l'*if*, ma si passa all'*else*. Per esempio 6 verifica la condizione poiché è divisibile sia per 2 che per 3; 8 non verifica la condizione poiché è divisibile per 2 ma non per 3.

||: è l'operatore "or" che significa oppure. Con questo operatore l'istruzione è eseguita se è verificata anche solo una delle condizioni. Esempio:

```
//Programma che verifica se un intero è divisibile per 2 oppure per 3
#include<iostream.h>
int main()
{
	int a;
	cout<<"Inserisci intero"<<endl;
	cin>>a;

	if(a%2==0 || a%3==0)
		cout<<"L'intero è un multiplo"<<endl;
	else
		cout<<"L'intero non è un multiplo"<<endl;

	return 0;
}
```

In questo caso basta che sia verificata una delle due condizioni e viene eseguito l'if. Per esempio il 6 da condizione vera, infatti è divisibile sia per 2 che per 3; 8 da condizione vera poiché è divisibile per 2; il 9 da vero poiché è divisibile per 3, mentre il 13 da condizione falsa poiché non è divisibile né per 2 né per 3.

!: è l'operatore "not" e serve per invertire il risultato di una condizione. Esempio:

```
//Programma che stabilisce se uno studente è bocciato
#include <iostream.h>
int main()
{
	int voto;
	cout<<"Inserisci il voto"<<endl;
	cin>>voto;

	if(!(voto>=18))
		cout<<"Lo studente è bocciato"<<endl;

	return 0;
}
```

L'operatore not di solito può essere evitato, comunque è bene saperlo usare.

Mettetevi alla prova

1- Scrivere un programma che legge 3 interi, verifica se costituiscono i lati di un triangolo e in caso affermativo scrive l'indicazione del tipo di triangolo. Suggerimenti: 1) in un triangolo ogni lato è minore della somma degli altri 2, ed è maggiore della loro differenza; 2) possono essere imposte condizioni del tipo "a>(b+c)".

Altri tipi di variabile

L'unico tipo di variabile che abbiamo incontrato sinora è stato *int*; in esso vengono memorizzati gli interi. Ovviamente esistono molti altri tipi di variabile, alcuni sono descritti di seguito.

- Char: il loro nome deriva da "character" e servono per gestire i caratteri. Si noti che in una variabile char può essere memorizzato un carattere solo. Ovviamente per dichiarali si scrive: *char nome_variabile;*
- Short: serve per gestire interi di piccole dimensioni(da -32767 a 32767)
- Long: serve per gestire interi di grandi dimensioni(da -2147483647 a 2147483647)
- Float: serve per gestire i numeri reali
- Double: gestisce numeri reali di grandi dimensioni

Switch

Le istruzioni *if* ed *if-else* sono delle istruzioni di selezione, dato che permettono di selezionare le azioni da eseguire ponendo delle particolari condizioni. L'istruzione "switch" è anche una istruzione di selezione, che permette di valutare più casi con un unico costrutto. Per capirci, serve per evitare di annidare una miriade di *if-else*. Vediamo come si usa con un esempio.

```
//Programma che trasforma un voto da italiano ad americano.
#include<iostream.h>
int main()
{
        int voto;
        cout<<"Inserisci un voto"<<endl;
        cin>>voto;

        switch(voto) //dentro la parentesi c'è la variabile, che è detta "espressione di controllo"
        {
                case 30: //dopo "case" deve essere inserito il possibile valore che può assumere la
                         //variabile
                case 29:
                case 28:
                case 27: cout<<"A"<<endl; //dopo i due punti deve essere
                                          //messa la(o le) istruzione da eseguire
                break;

                case 26:
                case 25:
                case 24: cout<<"B"<<endl;
                break;

                case 23:
```

```
            case 22:
            case 21: cout<<"C"<<endl;
            break;

            case 20:
            case 19:
            case 18: cout<<"D"<<endl;
            break;

            default: cout<<"F"<<endl;
            break;
        }

return 0;
}
```

Supponendo che l'utente inserisca un intero minore o uguale a 30 vediamo che:

1- Se il voto è 30, si ricade nel "case 30: " allora il programma vede l'istruzione da eseguire, la esegue e scende al caso dopo, esegue l'istruzione che c'è e scende, e così via sino al break. Nel nostro caso al case 30 non trova niente, al case 29 non trova niente, e così sino al case 27, dove trova un cout da eseguire, poi trova il break e si ferma.
2- Quando si arriva al break il programma esce dallo switch.
3- Default rappresenta l'else, infatti se il valore immesso non ricade nei casi studiati allora viene eseguita l'istruzione di default. Se non lo metto, il programma si comporta come un *if* senza l'*else*, ossia non esegue niente al di fuori dei casi studiati.

Mettetevi alla prova

Scrivere un programma che traduce i voti da americani ad italiani tenendo conto che:

1- "A" ed "a" equivalgono ad "ottimo"
2- "B" e "b" a "distinto"
3- "C" e "c" a "buono"
4- "D" e "d" a "sufficiente"
5- Il resto è insufficiente.

Ricapitolazione

Ora che abbiamo visto un bel po' di costrutti vediamo di eseguire un esercizio generale su quello che abbiamo fatto:

- Un numero che si legge allo stesso modo da sinistra e da destra è detto "palindromo", per esempio 12321 e 82928 sono palindromi. Scrivere un programma che legga un intero di cinque(5) cifre e determini se è palindromo. Suggerimento: utilizzare gli operatori modulo e divisione per separare il numero nelle singole cifre che lo compongono.

While

While è una istruzione di "iterazione", cioè serve ad eseguire più volte una operazione. In generale è strutturato in questo modo:

```
while(condizione)
{
        istruzione
```

```
}
```

la condizione è una relazione simile a quella già vista nei precedenti costrutti: se è soddisfatta viene eseguita l'istruzione, altrimenti viene saltata. Vediamo un esempio:

```
//programma che moltiplica 2 per se stesso finché è minore di 1000
#include<iostream.h>
int main()
{
        int prodotto=2;
        while(prodotto<=1000)
                prodotto*=2;

        return 0;
}
```

il programma si comporta in questo modo:

1- Crea la variabile "prodotto" e vi assegna il valore 2
2- Trovando un costrutto while verifica la condizione: prodotto, ossia 2, è minore di 1000? Si, quindi esegue "prodotto*=2"
3- Eseguita l'istruzione torna indietro alla condizione e verifica: prodotto, ora diventato 4, è minore di 1000? Si, quindi esegue ancora "prodotto*=2";
4- Ripete la stessa operazione sino a quando la condizione prodotto<=1000 diventa falsa. Allora salta l'istruzione ed esegue ciò che c'è dopo.

Col while si possono eseguire una gran quantità di problemi, con poche, ma semplici strategie.

Iterazione controllata da un contatore
L'iterazione controllata si esegue quando sappiamo a priori quante volte una certa operazione deve essere eseguita. Esempio:

```
//data una classe di 10 studenti vogliamo calcolare la media dei loro voti.
#include<iostream.h>
int main()
{
        int totale, contavoti, voto, media_voti;

        //fase di inizializzazione
        totale=0;
        contavoti=1;

        //fase di elaborazione
        while(contavoti<=10)
        {
                cout<<"Inserisci un voto"<<endl;
                cin>>voto;
                totale+=voto;
                contavoti++;
        }

        //fase finale
```

```
        media_voti=totale/10;
        cout<<"La media della classe è "<<media_voti<<endl;
        return 0;
}
```

il while viene eseguito per 10 volte, quindi si ferma quando contavoti diventa 11. La particolarità da notare sta nella variabile contavoti: come si intuisce è la variabile "contatore" ed è quella che inseriamo nella condizione del while; ogni volta che è eseguita una operazione questa viene incrementata, e quando arriva ad 11, il while non viene eseguito.
Le variabili "totale" e "contavoti" vengono sempre incrementate, pertanto bisogna prima inizializzarle, cioè assegnargli un valore.

Iterazione controllata da un valore sentinella.
Questo tipo di iterazione permette di eseguire un certo tipo di operazione sino a quando non viene immesso un determinato valore. Per esempio voglio fare la media dei voti di una classe, senza tuttavia conoscere il numero di componenti della classe. Per risolvere il problema impongo che vangano eseguiti i calcoli sino a quando non viene immesso il valore -1.

```
#include<iostream.h>
int main()
{
        int somma_voti, conta_voti, voto, media_voti;

        //fase di inizializzazione
        somma_voti=0;
        conta_voti=0;

        //fase di elaborazione
        cout<<"Inserisci il voto; digita -1 per terminare"<<endl;
        cin>>voto;
        while(voto!=-1)
        {
                somma_voti+=voto;
                conta_voti++;
                cout<<"Inserisci il voto; digita -1 per terminare"<<endl;
                cin>>voto;
        }

        //fase finale
        if(conta_voti!=0)
        {
                madia_voti=somma_voti/conta_voti;
                cout<<"La media della classe è "<< media_voti<<endl;
        }
        else
                cout<<"Non è stato immesso alcun voto"<<endl;
        return 0;
}
```

Si noti come il programma chiede di inserire nuovi valori sino a quando non viene inserito -1. Se volessimo essere precisi avremmo dovuto fare in modo che fossero immessi valori compresi tra 0 e 10.
A questo punto sorge un problema: se non posso servirmi di un valore sentinella? Allora si usa un valore di *EOF*(end of file), che si impone con una istruzione del tipo:

while(voto!=EOF)

e l'utente lo può usare premendo CTRL + Z, in windows, CTRL + D, in UNIX.

Mettetevi alla prova

1- Scrivere un programma che visualizza su schermo tutti gli interi, pari, tra 1 e 100.
2- Si legga una sequenza di interi e si determini: il numero degli interi pari, degli interi dispari e il numero totale dei valori inseriti. Si supponga che gli interi in gioco siano non negativi e che lo zero non sia né pari né dispari. La sequenza termina con un numero negativo.
3- Si crei un programma che calcola il fattoriale di un intero(si indica con !), dove per fattoriale si intende il prodotto di un intero per tutti i suoi interi precedenti escluso lo zero. Esempio: 5!=5*4*3*2*1=120, notate che il fattoriale di 0 e 1 è sempre 1.
4- Si crei un programma che chiede all'utente un intero n e visualizzi n elementi della serie di Fibonacci, che inizia con 0 1 e poi si forma così:
   0 1 1 2 3 5 8 13 ecc.
   quindi se l'utente digita 4 il programma deve mostrare i numeri 0 1 1 2

Booleane

Questo tipo di variabile è stata inventatala Bool e può assumere solo due valori: *true*(vero) e *false*(falso). In generale serve per gestire confronti e serve per verificare che dei valori godano di una certa proprietà. Vediamo un esempio.

```
//Programma che legge un intero e verifica se è primo
#include<iostream.h>
int main()
{
        int n; //numero da esaminare
        int j; //potenziale divisore
        bool e_primo; //questa è la nostra variabile booleana

        cout<<"Fornisci un intero positivo"<<endl;
        cin>>n;

        if((n == 1) || (n == 2))
                cout<<"Il numero è primo"<<endl;
        else
        {
                j=2;
                e_primo=true;
                while((e_primo == true) && (j<n))
                {
                        if(n%j == 0)
                        e_primo=false;
                        else
```

```
                j++;
        }

        if(e_primo==true)
                cout<<"Il numero è primo"<<endl;
        else
                cout<<"Il numero non è primo"<<endl;
    }

return 0;
}
```

vediamo come funziona il programma. Supponiamo che l'utente inserisca 33.

1- Il primo *if* viene verificato: il valore inserito è 1 oppure 2? No, quindi passiamo all'*else*
2- Viene imposto *j=2*, dove *j* è un divisore, ed *e_primo=true*.
3- C'è un *while*. Si verifica che *e_primo* è *true* e *j<n*? Si, quindi eseguiamo ciò che c'è nel *while*.
4- Nel *while* c'è per prima cosa un *if* da verificare: *n%j* è zero? Dato che *n=33* e *j=2* ciò non si verifica, quindi 2 non è un divisore di *n*.
5- Viene eseguito l'*else*, cioè *j* diventa 3.
6- Si ritorna alla condizione del *while*: anche ora che *j* è 3 la condizione è verificata.
7- Ora si verifica l'*if*: *33%3==0*? Si, quindi si esegue *e_primo=false*. Con questa istruzione la variabile cambia il suo valore. Ora si torna nella condizione del while.
8- È verificata la condizione? No. Quindi saltiamo l'istruzione.
9- Ora c'è un *if*, dato che *e_primo* è false, il numero non è primo.

Come vedete il while si ferma in 2 casi, o quando trova un divisore(come nel caso visto prima), oppure quando sono stati verificati tutti i possibili divisori(per cui *j<n*).

For

L'istruzione *for* è una istruzione di iterazione, quindi ha lo stesso scopo del while ed una struttura meno immediata ma più funzionale. Vediamo di fare un esempio.

```
//Esempio di iterazione controllata da contatore con un costrutto for
#include <iostream.h>
int main()
{
        for(int contatore=1; contatore<=10; contatore++)
                cout<<contatore<<endl;

        return 0;
}
```

Questo programma visualizza i numeri da 1 a 10. Nel *for* possiamo distinguere tre parti:

1- Nella prima vi è la variabile contatore, che può essere dichiarato nel *for* stesso
2- Nella seconda parte vi è la condizione da imporre per l'uscita dal *for*, esattamente come se dovessimo usare un while, questo significa che al suo interno ci possono stare booleane con operatori logici e comunque tutto quello che può servire a determinare una condizione di uscita

3- Infine c'è un incremento, anch'esso tipico di una iterazione con contatore, al posto di *contatore++* ci sarebbe anche potuto essere *contatore--* oppure *contatore+=2*, insomma, qualunque incremento sia necessario.

Si noti come non è sempre necessario riempire le tre parti del *for*, infatti possono esserci casi in cui l'incremento e la dichiarazione possono essere evitate. Allora dovremmo scrivere:

```
for( ; condizione; )
```

Un altro esempio di iterazione con il *for* è data da:

```
for(int i=7; i<=77; i+=7)
        cout<<i<<endl;
```

in questo caso verranno visualizzati tutti i multipli di 7 fino a 77.
Ancora possiamo fare:

```
for(int i=20; i>=2; i-=2)
        cout<<i<<endl;
```

in questo caso verranno visualizzati i numeri pari dal 20 al 2 in maniera decrescente.

Esercizio svolto

Di seguito viene esposto un esercizio svolto, nel quale verranno usati più o meno tutti gli operatori visti sino ad ora.

Programma che data una sequenza di interi positivi, terminante con -1, calcola la lunghezza di un sottosequenza(ossia il numero di interi di cui è composta) e calcola la somma dei numeri pari di tale sottosequenza e la mostra su video. Si noti che una sottosequenza termina con uno 0. Per esempio data la sequenza di:

2 5 7 6 0 5 7 3 4 2 0 1 16 -1

la prima sottosequenza ha lunghezza 4 e somma 8, la seconda ha lunghezza 5 e somma 6 e la terza lunghezza 2 e somma 17.

```
#include <iostream.h>
int main()
{
        int num, conta_l=0, somma_pari=0;

        cout<<"Inserisci un intero"<<endl;
        cin>>num;

        while(num<-1)
                cin>>num;
        //questo primo while evita che l'utente possa inserire interi minori di -1, ma potremmo
        //anche evitarlo

        while(num!=-1)
```

```
    {
        if(num==0)
        {
            cout<<"La lunghezza della sequenza è "<<conta_l<<endl;
            cout<<"La somma dei pari è "<<somma_pari<<endl;
            conta_l=0;
            somma_pari=0;
            cin>>num;
        }
        else if(num%2==0)
        {
            somma_pari+=num;
            conta_l++;
            cin>>num;
        }
        else
        {
            conta_l++;
            cin>>num;
        }
    }

    cout<<"La lunghezza della sequenza è "<<conta_l<<endl;
    cout<<"La somma dei pari è "<<somma_pari<<endl;

    return 0;
}
```

Num è la variabile in cui è memorizzato il valore immesso dall'utente, *conta_l* è un contatore che conta quanti sono i numeri inseriti, e *somma_pari…* è la variabile in cui sono memorizzate le somme dei numeri pari.

Mettetevi alla prova

1- Scrivere un programma che visualizza su schermo un quadrato di lato 5 e carattere #, cioè:

```
#####
#####
#####
#####
#####
```

2- Scrivere un programma che chiede all'utente di inserire un carattere C e un intero K e visualizzare un quadrato di lato K e carattere C.

3- Scrivere un programma che chiede all'utente un intero K, un carattere C e visualizzi un quadrato vuoto; se per esempio l'utente inserisce 3 e # deve essere visualizzato:

```
###
# #
###
```

4- Scrivere un programma che legge un intero dispari K e un carattere C e visualizza un triangolo; se per esempio l'utente inserisce 5 e # mostrare:

```
    #
   ###
  #####
```

5- Scrivere un programma che chiede all'utente un intero dispari K e un carattere C e visualizza un rombo con diagonale K, se per esempio viene inserito l'intero 7 e # deve essere visualizzato:

```
   #
  ###
 #####
#######
 #####
  ###
   #
```

6- Scrivere un programma che, letta una sequenza di interi, terminante con un numero negativo, determini il numero di 0, il numero di interi pari e di interi dispari inseriti.
7- Scrivere un programma che legga una sequenza di interi non nulli costituita da una serie di sottosequenze separate l'una dall'altra da 0 e calcoli e scriva su output la somma di ciascuna sottosequenza. Un numero negativo denota la fine dell'intera sequenza.
8- Leggere 2 interi A e B strettamente positivi e calcolare e scrivere il logaritmo intero di A in base B, ossia il più grande intero n tale che $B^n <= A$. Esempio, se A=10 e B=3, la risposta è 2 in quanto $3^2<10$ ma $3^3>10$.
9- Scrivere un programma che riceva una sequenza di dati relativi ai tassi di inquinamento registrati giornalmente in una certa località ed in un certo periodo di tempo e determini e stampi i picchi dell'inquinamento. Più in particolare, la sequenza è chiusa da un numero negativo. I tassi di inquinamento sono interi tra 0 e 100; un picco di inquinamento è un massimo relativo nella sequenza, ossia un dato che è maggiore del precedente e del seguente. Per ogni picco il programma deve visualizzare il valore del tasso di inquinamento ed il giorno in cui si verifica(i giorni vanno contati a partire da 1). Come consultivo il programma deve altresì visualizzare il numero di picchi rilevati ed il numero di dati da input. Prestare attenzione alla gestione del primo e ultimo dato.

Funzioni

I programmi che stiamo svolgendo in questo tutorial sono molto piccoli, tuttavia nella programmazione avanzata entrano in ballo anche migliaia di righe di codice. Per evitare di rendere il codice illeggibile si ha la buona abitudine di suddividere i programmi in "pezzettini", ossia creare dei sottoprogrammi che permettono di risolvere dei piccoli problemi all'interno di un grosso programma. Gli strumenti che permettono questo tipo di programmazione sono le funzioni e le classi. Noi ci occuperemo solo delle funzioni.

Vediamo un esempio di funzione, in cui si calcola il quadrato di un numero, che ovviamente verrà illustrato riga per riga.

```
//Esempio di uso di funzioni
#include<iostream.h>
int square(int); //prototipo di funzione
int main()
{
        for(int x=1; x<=10; x++)
                cout<<square(x)<<" ";
        cout<<endl;
        return 0;
}
//definizione di funzione
int square(int y) //y è un parametro di ingresso ed è una variabile
```

```
{
        int z;
        z=y*y;
        return z;
}
```

int square (int) è detto prototipo di funzione e serve a noi per dire al compilatore che useremo una funzione che restituisce un intero(questo è indicato dalla parole *int* **prima** di *square*), di nome *square*(che in inglese vuol dire quadrato), alla quale dovremo dare un parametro intero. Il codice non è scritto nel prototipo, ma dopo il main. Per capire ciò che avviene nel programma immaginiamo di essere il compilatore:

- Per prima cosa il compilatore legge ciò che c'è alla seconda riga, quindi si avvale della libreria <iostream.h>.
- Poi legge che nel programma principale verrà richiamato un sottoprogramma di nome *square*, il quale calcola un intero, e per fare questo calcolo ha bisogno di un altro intero.
- Ora inizia il main, ossia il programma principale.
- C'è un *for*, che inizializza la variabile *x* ad 1.
- Dentro il *for* c'è un *cout*, quindi deve essere mostrato qualcosa e questo "qualcosa" è *square(x)*.
- Quando il compilatore legge *square(x)* che cosa fa? Per prima cosa, avendo attribuito a *x* il valore 1 leggerà *"square(1)"*, poi ferma il main e passa alla funzione *square* in cui copia su *y* il valore di *x*, ossia 1. Questa funzione che cosa fa? Riceve un intero, che in questo caso è 1, e poi esegue i calcoli che vi abbiamo scritto. Nel nostro caso viene dichiarata una nuova variabile, detta *z*, e poi viene eseguito *z=y*y* quindi c'è *return z*, questo significa che al main verrà passato un valore uguale a *z*, ma quanto è *z*? È 1, dato che la *z* era *y*y=1*, quindi passerà al main il valore 1.
- Il main ora mostra su video 1, che è il risultato di *square*.
- Ora il *for* ha finito una operazione, quindi aumenta *x* di 1 e si ha x=2.
- Il compilatore legge *cout<<square(x)*, quindi passa alla funzione ed attribuisce alla *y* il valore di *x*.
- La funzione esegue *z=y*y* e ritorna 4
- Il main fa vedere su schermo 4
- Poi la *x* diventa 3, ed il procedimento viene ripetuto sino a *x=10*

A questo punto è indispensabile fare delle considerazioni:

- In generale una funzione ha questa struttura:

```
tipo_restituito nome_della_funzione (lista_dei_parametri)
{
        dichiarazioni
        istruzioni
}
```

- Può restituire un valore per volta e questo valore viene restituito per mezzo dell'istruzione "return" + il nome della variabile.
- Le variabili dichiarate in una funzione nascono e muoiono con essa.
- Il main è una particolare funzione, che non richiede niente(per questo ha le parentesi vuote) e non restituisce praticamente niente.

Facciamo un altro esempio per chiarire l'idea.

```
//esempio di funzione
#include <iostream.h>
int maximum (int, int, int);
int main()
{
	int a, b, c;

	cout<<"Inserisci 3 interi: ";
	cin>>a>>b>>c;

	cout<<"Il massimo tra i tre interi è: "<<maximum(a, b, c)<<endl;
	return 0;
}
int maximum(int x, int y, int z)
{
	int max=x;

	if(y>max)
		max=y;
	if(z>max)
		max=z;

	return max;
}
```

questo programma calcola il massimo tra tre interi. Si noti come il primo valore scritto in *maximum(a, b, c)* viene copiato in *x*, il secondo, cioè *b*, in *y* ed il terzo in *z*; questo significa che quando si scrive un programma si deve evitare di fare confusione e scrivere tutto con il massimo ordine.
In un programma ovviamente possono esserci più funzioni; vediamo un esempio:

```
//Programma che calcola il quadrato e il cubo
#include<iostream.h>
int quadrato (int);
int cubo (int);
int main()
{
	int numero, q, c;
	cout<<"Inserisci un intero: ";
	cin>>numero;

	q=quadrato(numero);
	c=cubo(numero);

	cout<<"Il suo quadrato è "<<q<<endl;
	cout<<"Il suo cubo è "<<c<<endl;

	return 0;
}
int quadrato(int a)
```

```
{
        return a*a;
}
int cubo(int b)
{
        return b*quadrato(b);
}
```

In questo programma abbiamo usato le funzioni per calcolare il valore di q e c. Notiamo come nel *return* possono essere inserite delle operazioni: in questi casi il compilatore restituisce al main il risultato di queste operazioni. Ancora si noti come nella funzione cubo abbiamo richiamato un'altra funzione: in questi casi il compilatore trova il risultato della seconda funzione chiamata e quindi lo usa.
Negli esempio che abbiamo visto sinora ci siamo serviti di funzioni che restituiscono interi; se volessimo restituire per esempio un valore true o false basta usare una funzione del tipo:

```
bool funzione(...)
```

ad una funzione è possibile passare anche delle booleane specificandolo tra parentesi, come abbiamo fatto per gli interi.

Mettetevi alla prova

1. Scrivere una funzione multiplo che riceve una copia di interi e determina se il secondo è multiplo del primo. La funzione riceve 2 interi e restituisce "true" se il secondo è multiplo del primo, false altrimenti. Si utilizzi questa funzione in un programma che legge da input una serie di coppie di interi.
2. Un numero intero è detto perfetto se esso è uguale alla somma dei suoi fattori, incluso 1 ed escluso il numero stesso. Per esempio 6 è perfetto poiché 6=1+2+3. Scrivere la funzione perfetto che determina se il numero passato come parametro è perfetto, quindi restituisce true. Si utilizzi questa funzione in un programma che determina tutti i numeri perfetti compresi tra 1 e 1000. Per ogni numero perfetto trovato, visualizzare anche i fattori.

Librerie

Abbiamo visto come con le funzioni è possibile risolvere dei piccoli problemi per volta. Nel programmare capita spesso di dover utilizzare ripetutamente delle funzioni simili. Tutti i problemi che hanno già una soluzione sono racchiusi nelle "librerie". Quindi le librerie sono dei "raccoglitori" di funzioni, ed infatti conoscendo bene le librerie è possibile risolvere una vasta gamma di problemi. Per utilizzare una libreria si usa il comando:

```
#include<nomelibreria>
```

Noi ci occuperemo solo della libreria matematica; per chiamarla si usa il comando *#include<math.h>*.
Una volta chiamata una libreria possiamo usare le sue funzioni. In quella matematica abbiamo:

- ceil(x): prende un numero reale x e restituisce l'intero immediatamente più grande.
- cos(x): prende un numero reale e restituisce il coseno.
- exp(x): restituisce e^x.
- fabs(x): prende un numero reale e da il valore assoluto.
- floor(x): prende un numero reale e da l'intero più piccolo.

- log(x): restituisce il logaritmo naturale di x.
- log10(x): restituisce il logaritmo in base 10 di x.
- pow(x, y): restituisce x^y.
- sin(x): restituisce il seno.
- sqrt(x): restituisce la radice quadrata di x.
- tan(x): restituisce la tangente di x.

Funzioni void

Si tratta di una funzione che non restituisce alcun valore al main, ed è strutturata come le altre funzioni già discusse. Un esempio particolare merita il caso in cui la lista dei parametri da passare alla funzione è vuota; vediamo subito.

```
//Esempio di funzione void
#include <iostream.h>
void funzione1( );
void funzione2( void );
int main( )
{
        funzione1( );
        funzione2( );
        return 0;
}
void funzione1( )
{
        cout<<"Questa funzione non riceve alcun argomento"<<endl;
}
void funzione2( void )
{
        cout<<"Anche questa funzione non riceve alcun parametro"<<endl;
}
```

Questo programma visualizza su schermo:

Questa funzione non riceve alcun argomento
Anche questa funzione non riceve alcun argomento

Oltre ciò che è stato detto si deve notare come le funzioni void non presentino il comando "return", questo proprio a sottolineare il fatto che le funzioni non restituiscano valori al main.
In questo momento il loro uso potrebbe apparire ininfluente, tuttavia le riprenderemo in seguito ed allora avremo una panoramica più ampia dell'argomento.

Funzioni in linea

Nella progettazione di un programma è utile servirsi delle funzioni, tuttavia è bene ricordare come ogni volta che una funzione viene richiamata viene impiegato un certo tempo di elaborazione. Per questo motivo le funzioni più piccole vengono scritte come funzioni in linea. Per utilizzarle si usa il comando *inline* ed hanno la particolarità che, piuttosto che essere richiamate, il loro codice viene copiato dove deve essere utilizzato.

```
//Calcolare il volume di un cubo
#include <iostream.h>
inline int cubo(int s){return s*s*s; }
```

```
int main( )
{
        int lato;
        cout<<"Inserisci il lato"<<endl;
        cin>>lato;
        cout<<"Il suo volume è "<<cubo(lato)<<endl;
        return 0;
}
```

E' immediato notare come la scrittura di questo programma va a tutto vantaggio della leggibilità.

Passaggio per valore e per riferimento

Quando abbiamo introdotto le funzioni abbiamo detto che in essa le variabili venivano copiate e quindi utilizzate per svariate operazioni. Anche se non lo abbiamo detto prima questo passaggio di variabile è detto "per valore", la sua particolarità sta nel fatto che anche se nella funzione la variabile viene modificata nel main il suo valore non viene compromesso. Il difetto del passaggio per valore è che per dati di grosse dimensioni si possono avere grandi quantità di memoria occupate. Un nuovo tipo di passaggio è il "passaggio per riferimento", con questo metodo la variabile non viene copiata bensì "collegata" alla funzione, in questo modo ogni modifica avvenuta nella funzione, si ripercuote anche nel main; è chiaro che con questo tipo di passaggio vi possono essere degli errori che anche il compilatore non è in grado di segnalare, per cui si faccia attenzione.

```
//Programma che, dati a e b, restituisce a^b e b^a
#include <iostream.h>
#include <math.h>
void potenze (int, int, int &, int &)
it main()
{
        int a, b, pot1, pot2;
        cout<<"Inserisci due numeri: ";
        cin>>a,b;
        potenza(a, b, pot1, pot2);
        cout<<"a^b= "<<pot1<<endl;
        cout<<"b^a= "<<pot2<<endl;
        return0;
}
void potenze(int x, int y, int &p1, int &p2)
{
        p1=pow(a, b);
        p2=pow(b, a);
}
```

Notiamo subito 2 cose.

1. Per passare le variabili per riferimento si usa il carattere & tra il tipo e il nome della variabile.
2. In una funzione possono essere passate variabili sia per valore sia per riferimento.

Inoltre se una variabile è passata per riferimento bisogna segnalarlo sia nella definizione della funzione che nella funzione vera e propria.
Per risolvere in modo corretto alcuni problemi è necessario usare il passaggio per riferimento. Vediamo degli esempi, il primo scorretto, il secondo giusto.

```
//Scambio fra due numeri-scorretto
#include<iostream.h>
void scambia (int, int);
int main()
{
        int a,b;
        a=3;
        b=4;
        cout<<"a= "<<a<<" b= "<<b<<endl;
        scambia(a, b);
        cout<<"a= "<<a<<" b= "<<b<<endl;
        return0;
}
void scambia(int x, int y)
{
        int t;
        t=x;
        x=y;
        y=t;
}


//Scambio di due numeri- corretto
#include<iostream.h>
void scambia(int &, int &)
int main()
{
        int a=3, b=4;
        cout<<"a= "<<a<<" b= "<<b<<endl;
        scambia(a, b);
        cout<<"a= "<<a<<" b= "<<b<<endl;
        return0;
}
void scambia(int &x, int &y)
{
        int t;
        t=x;
        x=y;
        y=t;
}
```

Si noti come nel primo esempio le variabili sono passate per valore e le modifiche non si ripercuotono nel main, mentre nel secondo esempio sono passate per riferimento ed i numeri sono effettivamente scambiati.

Mettetevi alla prova

1. Scrivere una funzione che riceve un argomento intero e restituisce il numero con le cifre al contrario. Per esempio se l'utente digita 12345, su schermo si leggerà 54321.

2. Scrivere una funzione “calcola” che riceve 2 variabili intere e restituisce la somma , la differenza e il prodotto. Nota: restituisce in questo caso non vuol dire stampa su schermo, ma passare al main!

Array

Sinora,nei nostri programmi, abbiamo sfruttato come unico strumenti di memorizzazione dei dati la variabile, e l’abbiamo descritta come una celletta in cui era possibile memorizzare un numero, un carattere ecc… L’array è una variabile formata da tante cellette *numerate* in cui memorizzare tanti dati dello *stesso* genere; il numero di cellette è la *dimensione* dell’array. Su ogni singola celletta è possibile effettuare tutte le operazioni che abbiamo incontrato. Vediamo un esempio in cui è inizializzato un array.

```
//Inizializzazione di un array
#include<iostream.h>
int main()
{
        int i, n[10];
        //inizializzazione
        for(i=0; i<0; i++)
                n[i]=0;
        cout<<”L’array è: “;
        for(i=0; i<10; i++)
                cout<<n[i]<<” “:
        cout<<endl;
        return0;
}
```

Ci sono un po’ di cose nuove da spiegare, perciò procediamo con ordine.

- Per dichiarare l’array si usa indicare prima il tipo di dato(nel nostro caso *int*), poi il nome dell’array(da noi “n”) ed infine tra parentesi quadre la dimensione dell’array(da noi 10). Nel dichiarare l’array è obbligatorio indicarne la dimensione.
- Nel lavorare con le variabili abbiamo detto che per effettuare operazioni su di esse avremmo potuto utilizzare espressioni come *a=1+2,* per gli array avviene una la stessa cosa, con la differenza che dobbiamo riferirci sempre ad un elemento particolare ed indicarlo per mezzo di un indice o di un numero, per esempio: *a[1]=3* o *a[x]=3*; con la prima scrittura indichiamo che all’elemento dell’array *a* con indice 1 assegniamo valore 3, mentre con la seconda espressione usiamo un indice, che assume valore maggiore o uguale a zero, per riferirci all’elemento, quindi se *x=0* si ha *a[o]=3*, quando *x=1* si ha *a[1]=3* e così via.
- Per inizializzare un array è necessario usare un *for* che permette di accedere ad un singolo elemento per volta. E’ necessario soffermarsi su una particolarità: la dimensione del nostro array è 10, però le celle non sono numerate da 1 a 10, bensì da 0 a 9, perciò il nostro indice(la variabile “i”) partirà da 0 ed arriverà a 9. Se il nostro indice arrivasse a 10 commetteremmo un errore, perciò evitiamo distrazioni.
- Per stampare su schermo un array è necessario un altro *for*, che permette di effettuare per ogni elemento un *cout*.
- All’interno delle parentesi quadre è possibile inserire anche delle espressioni come *a[i+j]=3* dove *i* e *j* hanno valori noti.

Per inizializzare un array è possibile usare una scrittura più semplice e veloce.

```
//Inizializzazione di un array nella dichiarazione
#include<iostream.h>
int main()
{
        int i;
        int n[10]={32, 27, 64, 18, 95, 14, 90, 70, 60, 37};
        cout<<"L'array è: ";
        for(i=0; i<10; i++)
                cout<<n[i]<<" ";
        cout<<endl;
        return 0;
}
```

Con questa scrittura abbiamo risparmiato un for, assegnando i valori come in figura.

32	27	64	18	95	14	90	70	60	37

In questo programma sono stati assegnati valori ben definiti a tutte le celle dell'array, comunque se avessimo scritto solo i primi 5 valori, il resto dell'array sarebbe stato inizializzato a 0. Notate come questa caratteristica risulti comoda nel caso in cui volessimo inizializzare un array, per esempio di dimensione 100 a zero, infatti basta usare una riga di codice:

```
int x[100]={0};
```

così al primo elemento diamo 0 ed il resto sarà 0 automaticamente.
Se per errore do più valori di quanti sono richiesti il compilatore mi da errore di sintassi.
Per inizializzare un qualunque array è utile usare una costante di supporto, in modo che successive modifiche al programma siano più rapide:

```
int main()
{
        const int dim=10;
        int n[dim];
        .
        .
        .
}
```

Per definire la dimensione bisogna servirsi sempre e solo di una costante.

Esempi

Di seguito riporto alcuni esempi di programmi che utilizzano gli array.

```
//Programma che somma gli elementi di un array di 12 elementi
#include<iostream.h>
int main()
{
        const int dim=12;
        int a[dim];
        int somma=0;

        cout<<"Inserisci un array di "<<dim<<" interi"<<endl;
        for(int i=0; i<dim; i++)
                cin>>a[i];
```

```
        for(int j=0; j<dim; j++)
                somma+=a[j];

        cout<<"La somma è "<<somma<<endl;
        return 0;
}

//Programma che costruisce un istogramma con gli asterischi pari al valore degli array
//Se per esempio nell'array ho 9 4 1, allora sarà visualizzato
//        *********
//        ****
//        *
#include<iostream.h>
int main()
{
        const int dim=10;
        int n[dim];

        for(int i=0; i<dim; i++)
                cin>>n[i];

        for(int j=0; j<dim; j++)
        {
                for(int z=0; z<n[j]; z++)
                        cout<<"*";
                cout<<endl;
        }

        return 0;
}
```

Mettetevi alla prova

- Leggere due array A e B di, al più, 10 elementi interi, calcolare il prodotto scalare e scriverlo su output. Si ricordi che il prodotto scalare è la somma dei prodotti delle componenti con lo stesso indice.
- Scrivere un programma che legge n interi e verifichi se esista un valore dell'insieme che è uguale alla somma di tutti gli altri.
- Si scriva un programma che legga una sequenza di 10 interi positivi, trovi il massimo ed emetta su output tale valore, il numero di volte che compare nella sequenza e le posizioni in essa occupate. Successivamente il programma deve trovare il submassimo e per esso deve emettere le stesse informazioni che sono state emesse per il massimo.
- Scrivere un programma che riceva in input 2 array di interi A e B e un intero K e indichi se ciascun elemento di A è divisibile per almeno K elementi di B.
- Scrivere un programma che legga due array di interi A e B; si supponga che i 2 array siano ordinati in modo crescente, il programma deve restituire un terzo array C ordinato in modo crescente che contenga tutti gli elementi di A e B con l'eventuale eliminazione dei duplicati.
- Un numero intero positivo molto grande, ad esempio un numero di 100 cifre, può essere rappresentato da un array di 100 elementi che contiene in ogni elemento una cifra dell'intero, in particolare nell'ultimo elemento c'è l'unità, nel penultimo la decina e così via; ovviamente in ogni elemento ci può essere un numero compreso tra 0 e 9. Si scriva un

programma che, dati 2 array A e B che rappresentano due interi della stessa dimensione, restituisca un terzo array C somma degli altri due. Si assuma che la somma di A e B non dia luogo a "trabocchi", pertanto C avrà la stessa dimensione di A e B. Si gestisca in maniera opportuna il riporto. Si operi in maniera analoga per la differenza D tra A e B. L'array D avrà le stesse dimensioni di A e B. Nota: se A[i] è minore di B[i] è necessario che A[i] si presti una decina da A[i-1] e nello stesso tempo A[i-1] va diminuito di 1.

- Scrivere un programma che riceve in ingresso 2 array di n interi a e b, e indichi in uscita se ciascun elemento di a è divisibile per almeno un elemento pari e almeno 2 elementi dispari di b.

Array e funzioni

Dello scopo delle funzioni ci siamo già occupati, pertanto per vedere come passare degli array ad una funzione ci serviremo di un esempio:

```
//Esempio di passaggio di un array ad una funzione
#include<iostream.h>
void modifica_array(int [], int);
void modifica_elemento(int);
int main()
{
	const int dim=5;
	int i, a[dim]={0, 1, 2, 3, 4};
	for(i=0; i<dim; i++)
		cout<<a[i];
	cout<<endl;

	modifica_array(a, dim);
	for(i=0; i<dim; i++)
		cout<<a[i];
	cout<<endl;

	cout<<a[3];
	modifica_elemento(a[3]);
	cout<<a[3]<<endl;

	return 0;
}
void modifica_array(int b[], int dim)
{
	for(int j=0; j<dim; j++)
		b[j]*=2;
}
void modifica_elemento(int e)
{
	e*=2;
}
```

Per passare gli array ad una funzione è necessario ricordare quanto segue:

- Anche se usiamo la scrittura tipica del passaggio per valore, gli array vengono passati sempre per riferimento. Il motivo di questa scelta sta nel fatto che gli array possono

occupare anche una gran quantità di memoria, perciò una loro copia potrebbe appesantire di molto l'esecuzione di un programma.
- La dimensione di un array è bene dichiararla accanto all'array stesso; si noti che se la dimensione è passata nelle parentesi quadre il compilatore la ignorerà.
- Nella funzione *modifica_elemento* abbiamo passato solo il singolo elemento dell'array; in questo caso il passaggio avviene per valore, pertanto, in questa funzione, in realtà non avviene alcuna modifica.

Mettetevi alla prova

- Scrivere una funzione che riceve un array di interi e un intero k e restituisce su schermo lo stesso array depurato da tutti i sottomultipli di k insieme alla nuova dimensione.
- Scrivere un funzione che riceve due array a e b di n interi e restituisca un array d che contiene tutti gli elementi di a che non sono presenti in b.
- Si scriva la funzione "compatta" che riceva un array di, al più, 100 elementi e provveda a restituire l'array "compattato", cioè senza duplicati. Si inserisca questo array in un programma che, tramite una funzione, legga l'array, poi avvii la funzione compatta, ed infine visualizzi l'array compattato.

Stringhe

Le stringhe sono degli array di caratteri, e ci serviranno, è ovvio, quando dobbiamo memorizzare parole. Per inizializzarle vanno bene i metodi visti sinora. Per dichiararli si può usare una scrittura del tipo:

char colore[]="blu";

la dichiarazione, se conosciamo a priori ciò che va messo, è più semplice rispetto agli altri tipi di array. La stringa che abbiamo dichiarato ha dimensione 4 poiché oltre ai caratteri che abbiamo deciso noi è memorizzato un carattere "\0". Perciò la nostra stringa è la seguente:

b	l	u	\0

Nell'inizializzare una stringa è meglio non essere troppo restrittivi con le dimensioni, poiché potrebbe non bastare lo spazio a disposizione.
Quando dobbiamo far inserire all'utente i caratteri possiamo scrivere direttamente:

```
char stringa[20];
cin>>stringa;
```

Allo stesso modo per visualizzarla:

```
cout<<stringa;
```

quindi il C++ ci risparmia di scrivere il *for*.
Per accedere ad un singolo elemento della stringa basta usare la scrittura tipica degli array:
stringa[3]
che significa che vogliamo accedere al terzo elemento della stringa
Ora vediamo un esempio:

```
//Programma sulle stringhe
#include<iostream.h>
int main()
{
        char stringa1[20], stringa2[]="Hello,world";
```

```
        cout<<"Inserisci una stringa: ";
        cin>>stringa1;
        cout<<"La stringa numero 1 è "<<stringa1<<endl;
        cout<<"La stringa numero 2 è"<<stringa2<<endl;
        cout<<"I singoli caratteri della stringa numero 2 sono: ";
        for(int i=0; stringa2[i]!='\0'; i++)
                cout<<stringa2[i]<<" ";
        cout<<endl;

        return 0;
}
```

Oltre a tutto quello che è stato già detto si noti come nel *for* la condizione di uscita è stata posta sapendo che nelle stringhe l'ultimo carattere è sempre \0 e gli apici servono poiché ci stiamo riferendo a dei caratteri.

Mettetevi alla prova

- Scrivere un programma che legge da input una parola lunga al più 30 caratteri e scriva su output un messaggio che indichi se la parola è palindroma oppure no. Si ricordi che è palindroma una parola che li legge identicamente da sinistra a destra e viceversa.

Ricerca e ordinamento

Capiterà spesso di scrivere funzioni che permettono di cercare dei particolari valori in un array. Vediamo un esempio di ricerca "lineare", cioè un ricerca fatta partendo dal primo ed arrivando all'ultimo elemento dell'array.

```
//funzione che restituisce true se esiste l'elemento k
bool ricerca_lineare(const int a[], int dim, int k)
{
        bool trovato=false;
        int i=0;
        while(i<dim && trovato==false)
        {
                if(a[i]==k)
                        trovato=true;
                else
                        i++;
        }
        return trovato;
}
```

Con questo metodo si vede se ogni elemento dell'array è quello cercato, se viene trovato la booleana diventa *true* e si esce dal while, invece se non viene trovato nulla si arriverà alla condizione *i=dim* e la booleana rimarrà *false*. Ecco ora un esempio in cui oltre a vedere se esiste, riportiamo anche la posizione dell'elemento cercato.

```
//Funzione che indica la posizione dell'elemento trovato
int ricerca_lineare(const int a[], int dim, int k)
{
        bool trovato=false;
        int i=0;
        while(i<dim && trovato==false)
```

```
            if(a[i]==k)
                trovato=true;
            else
                i++;
        if(trovato==true)
            return i;
        else
            return -1;
}
```

In questo caso abbiamo semplicemente riportato l'indice *i* per segnare la posizione dell'elemento cercato. Se non viene trovato niente si legge -1.
Se abbiamo un array ordinato si può usare un tipo di ricerca detta "binaria". Con questo metodo si parte dal centro dell'array, se l'elemento cercato è, per esempio, minore dell'elemento in mezzo, allora andrò nel centro della prima metà, se poi vedo che il mio elemento è maggiore di quest' ultimo allora andrò nel centro del secondo quarto, e così via sino a quando non trovo il mio elemento.

```
//Funzione ricerca binaria
int ricarca_binaria(int b[], int dim, int k)
{
        int iniziale, finale, centrale;
        iniziale=0;
        finale=dim-1;
        centrale=(iniziale+finale)/2;

        while(iniziale<= finale)
        {
            centrale=(iniziale+finale)/2;
            if(k==b[centrale])
                return centrale;
            else if(k<b[centrale])
                finale=centrale-1;
            else
                iniziale=centrale+1;
        }

        return -1;
}
```

Dopo la ricerca dobbiamo soffermarci sull'ordinamento; anche qui vi sono vari metodi.
Selection sort: con questo metodo si cerca il massimo di un array e lo si mette in ultima posizione(o in prima, dipende se l'ordinamento è crescente o decrescente). Vediamo un esempio:

```
//Funzione selection sort per ordinamento crescente
void selection_sort(int a[], int n)
{
        int j, imax, temp;
        for(j=n-1; j>=1; j--)
        {
            imax=0;
```

```
            for(int i=1; i<=j; i++)
            {
                if(a[i]>a[imax])
                    imax=i;
            }
            temp=a[imax];
            a[imax]=a[j];
            a[j]=temp;
        }
}
```

Con questo metodo partiamo dall'ultimo elemento e lo confrontiamo con tutti gli altri. L'indice *imax* si riferisce al valore più grande che abbiamo trovato. Una volta che è stato trovato con lo scambio fra due numeri viene portato il massimo in ultima posizione. Poi si passa al penultimo elemento e si ripetono le operazione di prima. Ed è così per tutti gli elementi.

Un secondo metodo è il "bubble sort": con questo metodo si parte dal primo elemento e lo si scambia con quello dopo se è più grande, scorrendo tutto l'array il più grande viene portato in ultima posizione. Il nome deriva dal fatto che i numeri sono come "bolle" che salgono di posizione se è verificata la condizione. Vediamo l'esempio:

```
//Funzione bubble_sort
void bubble_sort(int a[], int n)
{
        int j, temp, i;
        for(j=n-1; j>=1; j--)
        {
            for(i=0; i<=j-i; i++)
                if(a[i]>a[i+1])
                {
                    temp=a[i];
                    a[i]=a[i+1];
                    a[i+1]=temp;
                }
        }
}
```

Matrici

La matrice è un array di 2 indici; per scorrerlo è possibile rappresentare la matrice come una tabella in cui il primo indice rappresenta la riga ed il secondo indice è la colonna. In generale con C++ posso sfruttare un array con 12 indici. Consideriamo per esempio una matrice 3x4; per dichiararla si indica con:

```
int m[3][4];
```

ed è una tabella del genere:

	Colonna 0	**Colonna 1**	**Colonna 2**	**Colonna 3**
Riga 0	m[0][0]	m[0][1]	m[0][2]	m[0][3]

Riga 1	m[1][0]	m[1][1]	m[1][2]	m[1][3]
Riga 2	m[2][0]	m[2][1]	m[2][2]	m[2][3]

Nella tabella ho segnato gli indici con cui identificare gli elementi all'interno della matrice. Notate come anche qui le righe e le colonne vengono contate da zero. Essendo degli array, le matrici conservano le loro proprietà nel passaggio alle funzioni.
Per inizializzare una matrice si può usare una scrittura del tipo:

int b[2][2]={{1, 2}, {3, 4}};

gli elementi tra le parentesi graffe interne rappresentano elementi della stessa riga, perciò la matrice b è la seguente:

1	*2*
3	*4*

Dato che valgono le stesse regole degli array se scrivo:

int b[2][2]={{2}, {3, 4}};

ottengo:

2	0
3	4

Ancora posso scrivere:

int b[2][2]={1, 2, 3};

da cui:

1	2
3	0

Adesso vediamo un programma in cui vengono stampate 2 matrici; si notino i commenti.

```
//Esempio
#include<iostream.h>
void stampa_matrice(int [][3]); //la colonna deve essere dichiarate nella parentesi
int main()
{
        int array1[2][3]={{1, 2, 3}, {4, 5, 6}};
        int array2[2][3]={1, 2, 3, 4, 5};
        int array3[2][3]={{1, 2}, {4}};
        cout<<"La prima matrice è: "<<endl;
        stampa_matrice(array1);
        cout<<"La seconda matrice è: "<<endl;
        stampa_matrice(array2);
        cout<<"La terza matrice è: "<<endl;
        stampa_matrice(array3);
        return 0;
}
```

```
void stampa_matrice(int a[][3])
{
	for(int i=0; i<2; i++)
	{
		for(int j=0; j<3; j++)
			cout<<a[i][j]<<" ";
		cout<<endl;
	}
}
```

Scorrere una matrice è come scorrere un array, con la differenza che dobbiamo usa due for per gestire 2 indici. Si faccia attenzione anche alle condizioni imposte nei for.
E se volessimo effettuare una somma? Tralasciando la parte iniziale, ecco come fare:

```
//Somma degli elementi
.
.
.
totale=0;
for(int x=0; x<3; x++)
	for(int y=0; y<4; y++)
		totale+=a[x][y];
```

Adesso vediamo un esempio in cui usiamo matrici e funzioni per determinare la media, in massimo e il minimo dei voti di una classe di studenti.

```
//Esempio sulle matrici
#include<iostream.h>

const int studenti=3;
const int esami=4;

int minimo(int [][esami], int, int);
int massimo(int [][esami], int, int);
float media(int [], int);
void leggi_matrice(int [][esami], int, int);
void stampa_matrice(int [][esami], int, int);

int main()
{
	int voti_studenti[studenti][esami] ;

	leggi_matrice(voti_studenti, studenti, esami);
	stampa_matrice(voti_studenti, studenti, esami);

	cout<<"Il voto massimo è "<<massimo(voti_studenti, studenti, esami)<<endl;

	cout<<"Il voto minimo è "<<minimo(voti_studenti, studenti, esami)<<end;

	for(int persona=0; persona<studenti; persona++)
	{
```

```
            cout<<"Il voto medio dello studente "<<persona<<" è ";
            cout<<media(voti_studenti[persona], esami)<<endl;
        }

        return 0;
}
void leggi_matrice(int voti[][esami], int s, int e)
{
        cout<<"Inserisci i voti"<<endl;

        for(int i=0; i<s; i++)
            for(int j=0; j<e; j++)
                cin>>voti[i][j];
}
void stampa_matrice(int voti[][esami], int s, int e)
{
        cout<<"I voti degli studenti sono "<<endl;

        for(int i=0; i<s; i++)
        {
            for(int j=0; j<e; j++)
                cout<<voti[i][j];
            cout<<endl;
        }
}
int minimo(int voti[][esami], int s, int e)
{
        int minimo_corr=voti[0][0];

        for(int i=0; i<s; i+)
            for(int j=0; j<e; j++)
                if(voti[i][j]<minimo_corr)
                    minimo_corr=voti[i][j];

        return minimo_corr;
}
int massimo(int voti[][esami], int s, int e)
{
        int max=voti[0][0];

        for(int i=0; i<s; i++)
            for(int j=0; j<e; j++)
                if(voti[i][j]>max)
                    max=voti[i][j];

        return max;
}
float media(int voti_stud[], int e)
{
        int totale=0;
```

```
        for(int i=0; i<e; i++)
                totale+=voti_stud[i];

        return (float)totale/e;  //col float il numero intero diventa reale
}
```

Questo programma è più lungo di quelli visti sinora, ma non ci sono molte particolarità da segnalare, al più il modo in cui ho dichiarato le costanti "studenti" ed "esami": le ho dichiarate fuori dal main in modo che siano riconosciute in tutte le parti del programma; le variabili dichiarate in questo modo si dice che hanno una *visibilità a livello di file*.

Scorrere una matrice

Dato che l'utilizzo di una matrice è simile a quello di un array, ciò che può creare qualche problema è scorrere la matrice, cioè prendere dei particolari valori che risiedono in particolari parti della matrice. Prima di procedere con un esempio voglio ricordare che una matrice si dice quadrata se il numero di righe è uguale a quello delle colonne.
Abbiamo la matrice quadrata come in figura, in cui possiamo distinguere la parte nord, sud, ovest ed est. Vogliamo una funzione(tralasciamo il main) che restituisce true se la somma degli elementi della parte nord è uguale alla somma degli elementi della parte ovest, senza considerare gli elementi della diagonale.

*								*
	*						*	
		*				*		
			*		*			
				*				
			*		*			
		*				*		
	*						*	
*								*

```
#include<iostream.h>
const int n=9;
bool matrice(int mat[][n]; int dim)
{
        int i, j, somma_nord=0, somma_ovest=0, centro=n/2;

        for(i=0; i<centro; i++)
                for(j=i+1; j<=dim-2-i; j++)
                        somma_nord+=mat[i][j];

        for(j=0; j<centro; j++)
                for(i=j+1; i<=dim-2-j; i++)
                        somma_ovest+=mat[i][j];

        return (somma_nord==somma_ovest); //se la condizione è verificata si restituisce true
}
```

Si nota come la difficoltà sta nel trovare la legge da inserire nelle condizioni del for. Un metodo per ricavarla è quello di scrivere in colonna le coordinate di tutti gli elementi che entrano in gioco(oppure quelli che stanno agli estremi) e vedere, per tentativi, il modo in cui variano.

Per esempio per gli elementi a nord dell'esercizio precedente si ha:

Riga	Colonna
0	1
1	2
2	3
3	4
2	5
1	6
0	7

Prima di tutto sappiamo che la riga varia da zero alla metà della matrice(n/2), poi vediamo dalla tabella come ogni riga parte da un indice che è più grande di 1 rispetto all'indice della riga(j=i+1) e finisce nella colonna che è *dim-2-i*. Questa ultima condizione potrebbe sembrare poco immediata, ma riflettendoci un po' non sarà difficile ricavare le leggi.

Mettetevi alla prova

In questi esercizi si trascuri il main e si scrivano sono le funzioni.

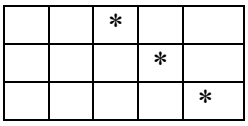
- Scrivere una funzione che ricevuta una matrice quadrata di interi restituisce true se la somma degli elementi a nord è uguale al prodotto degli elementi a sud, e contemporaneamente la somma degli elementi a ovest è uguale al prodotto degli elementi a est. Le diagonali non devono essere considerate.
- Due matrici quadrate di interi a e b, quadrate, si definiscono triangolarmente accoppiate se, indicati con a1, b1, a2, b2 i triangoli di figura sotto si ha che la somma degli elementi di a1 è uguale al prodotto degli elementi di b1 e il prodotto degli elementi di a2 è uguale alla somma degli elementi di b2. Si assuma int tutto ciò che a1, a2, b1, b2 comprendano anche le diagonali da cui sono limitate. Si scriva una funzione t_accoppiate che, ricevute in input 2 matrici a e b di dimensione n*n restituisca true, se queste sono triangolarmente accoppiate, false altrimenti.(Nota: in figura ho indicato la zona cui ci si deve riferire e gli asterischi rappresentano il limite massimo).

*				
	*			
		*		
			*	
a1				*

				*
			*	
		*		
	*			
*				b1

a2				*
			*	
		*		
	*			
*				

*				b2
	*			



Note conclusive

Con le matrici il tutorial giunge al termine, per chi ha voglia di andare avanti il passo successivo è quello di conoscere i puntatori, le strutture sino ad arrivare a padroneggiare le classi.

Se avete suggerimenti, critiche insomma qualunque cosa che possa far migliorare questo tutorial, non esitate a contattarmi per e-mail: link01_l@hotmail.com