Puntatori e variabili dinamiche

I puntatori sono un concetto molto importante nell'ambito della programmazione, ma al contempo sono abbastanza difficili da capire (soprattutto all'inizio) e da utilizzare correttamente.

Un **puntatore** è un particolare tipo di dato che serve a lavorare con gli indirizzi in memoria delle variabili. In generale, anche le variabili di tipo puntatore si chiamano puntatori a loro volta.

In C++ esistono sia i puntatori generici (di tipo `void *`), sia i puntatori tipizzati. In queste dispense considereremo solo questi ultimi.

Se T è un tipo di dato, allora una variabile dichiarata di tipo T* è un puntatore a T.

Ad esempio

```
int *p1;
```

```
double *p2;
```

p1 è un puntatore a int e *p2* un puntatore a double, ovvero *p1* può contenere indirizzi di variabili di tipo int e *p2* indirizzi di variabili di tipo double.

L'indirizzo di memoria associato ad una variabile si estrae con l'operatore &. Perciò

```
int m=10;
```

```
double x=3.1;
```

```
p1=&m;
```

```
p2=&x;
```

sono assegnamenti corretti: *p1* conterrà l'indirizzo di *m* e *p2* quello di *x*.

Si dice che *p1* "punta" a *m* e *p2* "punta" a *x*.

Invece l'assegnamento `p1=&x` è scorretto in quanto *x* è una variabile di tipo double e il suo indirizzo non può essere memorizzato in un puntatore di tipo int.

Se *p* è un puntatore che contiene l'indirizzo di una variabile *v*, allora la particolare notazione **p* permette di accedere al valore di *v* in un'espressione, sia di modificarlo tramite assegnamento. Detto in altri termini, *p è come se fosse un altro nome (*alias*) per *v*.

Per cui

```
b=(*p)+1;
```

assegna a *b* il valore di *m+1*, cioè 11, mentre

```
*p=17;
```

assegna a *m* il valore 17.

E' possibile assegnare il contenuto di un puntatore ad un altro (dello stesso tipo).

Ad esempio con

```
int a;
int *p, *q;
p=&a;
q=p;
```

entrambi i puntatori *p* e *q* conterranno l'indirizzo di *a*, quindi *p e *q sono sostanzialmente sinonimi. Infatti

```
*p=12;
cout << *q << endl;
```

scriverà sullo schermo 12.

Un puntatore può contenere il valore particolare 0 (o NULL in linguaggio C, o `nullptr` nelle nuove versioni di C++), che indica che il puntatore non è collegato a nessuna variabile. Infatti 0 non è considerato un indirizzo valido. Se *p* è un puntatore non inizializzato o contenente il valore 0, allora l'uso di *p è scorretto e può condurre a situazioni analoghe a quelle viste per gli indici errati in un array (errore di “Segmentation fault” o malfunzionamenti).

Allocazione dinamica

Uno degli scopi principali per l’utilizzo dei puntatori è la gestione delle variabili **allocate dinamicamente**.

L'**allocazione** è la modalità con cui viene assegnata una zona di memoria ad una variabile al momento della sua creazione, mentre la **deallocazione** è il processo inverso, con cui la memoria viene restituita al sistema quando tale variabile cessa di esistere.

Le variabili locali sono allocate su una parte di RAM chiamata “stack”. Lo stack permette di deallocare velocemente le variabili nell'ordine inverso con cui sono state allocate e ciò ha senso in virtù del meccanismo di creazione ed eliminazione delle variabili nelle funzioni e, più in generale, nei blocchi.

Le variabili **dinamiche** sono invece allocate e deallocate dal programmatore con i comandi **new** e **delete**, rispettivamente. Poiché il programmatore può deallocare le variabili seguendo un ordine qualsiasi, la parte di RAM usata per contenere le variabili dinamiche deve essere gestita in un modo diverso dallo stack. La zona di memoria in questione si chiama “heap”.

L'operatore new ha la sintassi

new *tipo_di_dato*

Esso alloca una variabile dinamica del tipo di dato indicato e restituisce come risultato l'indirizzo associato alla variabile. Di solito il risultato di new è memorizzato in un puntatore.

Ad esempio

```
int *p;
p=new int;
```

L’ultima istruzione crea una nuova variabile dinamica e ne memorizza l’indirizzo in *p*. A questo punto, *p si riferisce alla variabile dinamica appena allocata e può essere usato “come se” fosse una variabile di tipo int.

L'operatore delete ha la sintassi

delete *puntatore*

Esso dealloca la variabile dinamica il cui indirizzo è memorizzato nel puntatore indicato. Da quel momento in poi, la variabile dinamica non è più utilizzabile e lo spazio di memoria ad essa concesso è restituito al sistema, che potrà riutilizzarlo in seguito per altri scopi. Il puntatore comunque rimane inalterato e continuerà a conservare il vecchio indirizzo della variabile.

Le variabili dinamiche possono essere utilizzate come le variabili “normali” (ovvero quelle allocate automaticamente o staticamente). L’unica differenza “sintattica” è che le variabili dinamiche non hanno nome e possono essere usate solo tramite puntatori.

```
int p*=new int;
*p=40;
int a=(*p) * 2;
cout << *p << endl;
delete p;
```

Le variabili dinamiche però possono essere create in una funzione e poi utilizzate e distrutte in altre funzioni: in pratica la gestione dinamica va al di là del rigido schema “a blocchi” delle variabili locali, senza però degenerare nelle variabili globali, che invece sono allocate in memoria per tutto il corso del programma.

Le variabili dinamiche sono poi alla base delle strutture dati dinamiche (si vedano gli esercizi 5-8 e le note implementative del capitolo relativo a STL).

Un errore molto comune nella gestione delle variabili dinamiche è di tentare di utilizzare una variabile anche dopo che è stata deallocata. Questa situazione si può verificare, ad esempio, se l'indirizzo di una variabile dinamica è memorizzato in due (o più) puntatori

```
int *p1=new int;
....
int *p2=p1;
...
delete p1;
...
```

```
*p2=3;
```

In questo esempio la variabile dinamica è unica, ma poiché la variabile è stata eliminata tramite *p1*, il programmatore continua ad usarla tramite *p2*, così facendo si ottiene un errore di tipo “Segmentation fault” o malfunzionamenti del programma.

Gestione dello heap

Per capire meglio il funzionamento di new e delete, è utile vedere come è organizzato lo heap. Esso è diviso in parti libere e parti occupate. All'inizio vi è un'unica parte libera, che occupa tutto lo heap, e non vi sono parti occupate.

L'operatore *new* cerca una parte libera P grande a sufficienza da contenere il tipo di dato richiesto, supponiamo di B byte. La parte P viene quindi assegnata alla nuova variabile dinamica. Se P è grande esattamente B byte, P è occupata tutta. Se invece serve uno spazio minore, P viene divisa in due parti: una porzione di B byte viene occupata, mentre la porzione rimanente resta libera.

L'operatore *delete* non fa altro che dichiarare che la parte di heap associata alla variabile dinamica torna ad essere libera. A regime, nello heap si alternano parti libere e parti occupate, in cui alcune parti libere sono state precedentemente occupate con new e poi restituite con delete.

Puntatori e array

Oltre che il normale impiego, è possibile associare ad un puntatore di tipo T l’indirizzo di un elemento di un array di tipo T.

Ad esempio, in

```
int *p;
int a[5]={6,7,10,11,3};
p=&a[2];
```

il puntatore *p* punta al terzo elemento dell’array (il cui contenuto è 10).

E’ possibile usare gli indici anche con un puntatore *p*: questa operazione ha senso solo se *p*, come nell’esempio precedente, punta ad un elemento di un array.

La notazione p[i] indica l’accesso alla i-esima cella dell’array partendo da quella puntata da p.

Quindi p[0] corrisponde ad a[2], p[1] ad a[3], p[2] ad a[4], mentre p[-1] corrisponde ad a[1] e p[-2] ad a[0].

Ad esempio

```
cout << p[2] << endl;
```

stampa sullo schermo 3, mentre

```
p[-1]=5;
```

assegna ad a[1] il valore 5.

Ovviamente p[3] non esiste (corrisponderebbe all'elemento a[5]) e nemmeno p[-3].

Al posto di scrivere

```
p=&a[0];
```

si può scrivere semplicemente

```
p=a;
```

Infatti il nome di un array può essere visto come un puntatore che punta al primo elemento dell'array, ovvero *a* è equivalente ad *&a[0]*.

Questo fatto può essere usato per spiegare come avviene il passaggio dei parametri di tipo array: il parametro è tutti gli effetti un puntatore e l'argomento corrispondente è l'indirizzo del primo elemento. Infatti è possibile anche usare un puntatore al posto di un parametro di tipo array.

Ad esempio

```
double media_array(double *a,int n) {
     double somma=0;
     for(int i=0; i<n; ++i)
          somma=somma+a[i];
     return somma/n;
}


int main() {
     double arr[5]={7,8,10,4,3};
     double m=media_array(arr,5);
     cout << m << endl;
}
```

In C++ è anche possibile compiere operazioni aritmetiche con i puntatori: sommare ad un puntatore un numero intero e sottrarre due puntatori dello stesso tipo. Tali operazioni non saranno trattate in queste dispense.

Array dinamici

In C++ è possibile creare un array dinamico con

new *tipo_di_dato* [*dimensione*]

in cui *dimensione* è un'espressione intera che indica il numero di elementi del nuovo array.

Il risultato di new è l'indirizzo del primo elemento dell'array e può essere memorizzato in un puntatore del tipo degli elementi. Inoltre è possibile accedere agli elementi dell'array dinamico tramite il puntatore e l'indice, come si è visto nella sezione precedente.

Ad esempio

```
int n=10;
int *v=new int[n];
for(int i=0; i<n; ++i)
     v[i]=i*i;
int a=v[3];
```

Un array dinamico può essere cancellato con una versione particolare di delete:

```
delete[] v
```

Una matrice dinamica può essere creata utilizzando un array dinamico di puntatori, ognuno dei quali contiene l'indirizzo della riga corrispondente, che è sua volta un array dinamico di elementi.

Ad esempio

```
int nr=5,nc=3;
double **matrice;
matrice=new double*[nr];
for(int i=0; i<nr; ++i)
     matrice[i]=new double[nc];
```

La matrice dinamica così allocata è una normale matrice “rettangolare”, in cui ogni riga ha lo stesso numero di elementi, ma nulla vieta di creare matrici “jagged”, in cui ogni riga potrebbe avere un numero diverso di elementi. Ad esempio è possibile creare una matrice triangolare o una matrice tridiagonale.

Una matrice dinamica può essere usata con il doppio indice, esattamente come le matrici normali. Inoltre può essere passata ad una funzione senza incorrere nei problemi visti nel passaggio delle matrici.

Per cui si può definire

```
void scrivi_matrice(double **matrice,
          int num_righe, int num_colonne) {
…
}
```

Puntatore a strutture

Un puntatore ad una struct può essere dichiarato con lo stesso modo con cui si definisce un puntatore ad un altro tipo tipo di dato

```
struct punto {
     double x,y,z;
};
punto *p=new punto;
```

Invece di usare la notazione

```
(*p).x=3;
```

si può usare una notazione più leggera

```
p->x=3;
```

In generale, -> si usa con i puntatori a struct, nello stesso modo in cui si usa il punto con le variabili di tipo struct.

Puntatori e passaggio per riferimento

I puntatori sono alla base del funzionamento del passaggio per riferimento. Infatti in

```
void raddoppia(int x,int &y) {
     y=2*x;
}

int main() {
     int a;
     raddoppia(5,a);
     cout << a << endl;
}
```

quello che “realmente” accade è che il compilatore traduce il programma come se fosse stato scritto così

```
void raddoppia(int x,int *y) {
     *y=2*x;
```

```
}


int main() {
     int a;
     raddoppia(5,&a);
     cout << a << endl;
}
```

Ovvero

- il parametro *y* viene visto come se fosse un puntatore, nella fattispecie ad int;
- alla funzione prova viene passato l'indirizzo della variabile *a* che viene copiato in y;
- l'assegnamento a y viene considerato come assegnamento a *y.

In realtà, la dichiarazione `int &y` significa y è un “riferimento” ad una variabile int.

Un riferimento è un tipo di dato molto simile ad un puntatore, infatti entrambi contengono l’indirizzo di una variabile. Le differenze principali tra puntatori e riferimenti sono due.

Innanzitutto un riferimento *r* può essere associato ad una variabile solo durante la sua dichiarazione, tramite un’inizializzazione

```
int a;
a=17;
int &r=a;
```

ma non può essere successivamente associato ad un’altra variabile.

Inoltre ogni volta che si usa *r*, sia in un assegnamento che in un’espressione, si intende accedere alla variabile associata ad *r* (e non a *r* stesso).

Ad esempio

```
cout << r << endl;
```

stampa il valore 17, mentre

```
r=r+1;
```

fa diventare 18 il valore di *a*.

In pratica un riferimento è un puntatore “costante”, ovvero l’indirizzo contenuto nel riferimento non cambia.

Puntatori e riferimenti a costanti

Leggermente diverso è il caso di puntatori a costanti, ovvero puntatori a dati che non possono essere modificati.

Ad esempio

```
int a=5;
const int *p=&a;
cout << *p << endl;
```

è valido e scrive 5 sullo schermo, mentre

```
int a=5;
const int *p=&a;
*p=7;
```

non è accettato perché si sta tentando di modificare la variabile puntata da p.

Allo stesso modo è possibile definire un riferimento a costante

```
int a=5;
const int& p=a;
cout << p << endl;
```

ma p non piuò essere modificato.

Il passaggio per riferimento (a) costante può essere utile se si vuole passare un parametro per riferimento, ma tale parametro non deve essere modificato (e quindi anche il corrispondente argomento).

Esercizi

1. Scrivere un esempio di programma che crea alcune variabili “normali” e vi accede tramite puntatori

2. Scrivere una funzione che dati gli indirizzi di due variabili int (passati in due puntatori a int p1 e p2), scambia i valori delle due variabili

3. Scrivere una funzione che dati tre numeri reali a,b,c e due puntatori a numeri reali p1 e p2, memorizza nelle due variabili puntate da p1 e p2 le due soluzioni dell’equazione di secondo grado $ax^2+bx+c=0$. Se il delta è negativo, non si deve memorizzare alcun dato.

4. Scrivere una funzione che dati due numeri interi m e n (con m<=n), restituisce come risultato un array dinamico contenente tutti i valori compresi tra m e n.

5. Definire una struct *nodo* contenente un campo *key* di tipo int e un campo *next* di tipo puntatore a *nodo*. In pratica ogni nodo contiene l'indirizzo del nodo "successivo". I nodi possono formare una sorta di "catena", chiamata "lista". Definire una funzione *crea_nodo* che, dato l'indirizzo di un nodo n e un valore intero k, crea un nuovo nodo che ha come chiave k e come next il valore n, restituendo come risultato l'indirizzo del nuovo nodo.

6. Ponendo che l'ultimo elemento di una lista ha come valore 0 per il campo next, creare la lista 1-→2-→3-→4, memorizzando nella variabile *t* l'indirizzo del primo nodo, usando la funzione crea_nodo.

7. Definire una funzione che dato l'indirizzo del primo nodo di una lista (creata come nell'esercizio 6), calcola il numero dei nodi presenti nella lista.

8. Definire una funzione che dato l'indirizzo del primo nodo di una lista (creata come nell'esercizio 6), scrive sullo schermo le chiavi dei nodi presenti della lista.

La libreria STL

Il linguaggio C++ ha solo due forme di aggregazioni di dati, gli array e le struct. In particolare, gli array consentono di gestire collezioni di dati dello stesso tipo che possono essere utili in moltissime applicazioni. Gli array però costituiscono una struttura dati di tipo statico, in cui il numero degli elementi è fissato a priori e non sono efficienti le comuni operazioni di inserimento, cancellazione e ricerca.

Per ovviare a questi problemi, la libreria STL (Standard Template Library) fornisce una serie di strutture dati dinamiche già predefinite

- vector
- list
- queue, deque, stack
- priority_queue
- map e multimap
- set e multiset

che sono implementate in modo efficiente e sono semplici da utilizzare, grazie alla programmazione orientata agli oggetti e ai “template”.

I vector

I **vector** sono array gestiti dinamicamente in modo automatico e trasparente per il programmatore.

Un vector è una collezione di dati tutti dello stesso tipo T ed è composto da un campo, che contiene l’indirizzo di un array dinamico di tipo T, e da altri due campi distinti che rappresentano il “numero di elementi”:

- la **dimensione**, ovvero il numero effettivo di elementi utilizzati nel vector;
- la **capacità**, ovvero il numero massimo di elementi che si possono memorizzare nel vector.

In pratica l’array dinamico gestito dal vector ha dimensione pari alla capacità, ma il numero di elementi presenti (la dimensione) è minore o uguale alla capacità. Il vector gestisce in modo ottimale la capacità, aumentandola o diminuendola opportunamente quando è necessario.

Per usare i vector bisogna includere

```
#include <vector>
```

Al momento della creazione di un vector si può specificare una dimensione iniziale, oppure creare un vector inizialmente vuoto.

Una variabile di tipo vector si dichiara indicando il tipo degli elementi tra parentesi angolari

```
vector<int> v;
```

oppure indicando anche il numero iniziale di elementi

```
vector<int> v(10);
```

Nel primo caso *v* è inizialmente vuoto, nel secondo caso ha 10 elementi (come dimensione).

In ogni caso, all'inizio la capacità è uguale alla dimensione.

Si possono anche creare vector di tipi diversi, ad esempio

```
vector<double> v1;
vector<string> v2;
```

Il tipo di dato vector è tecnicamente una classe "template", ovvero una classe parametrica rispetto ad uno o più tipi di dato. Non approfondiremo ulteriormente tale argomento in queste dispense.

Un vector può essere usato come un array, ad esempio

```
b=v[3];
v[4]=11;
```

o

```
sum=0;
for(int i=0; i<10; ++i)
     sum=sum+v[i];
```

Inoltre è possibile eseguire le seguenti operazioni "veloci" (cioè che richiedono un tempo di esecuzione indipendente dalla dimensione di v)

- v.size(), restituisce la dimensione di v
- v.push_back(x), aggiunge x in fondo a v
- v.pop_back(), elimina l'ultimo elemento di v
- v.front(), restituisce il primo elemento (equivale a v[0])
- v.back(), restituisce l'ultimo elemento (equivale a v[v.size()-1])
- v.clear(), svuota v.

In particolare, l'operazione di push_back aggiunge un elemento in fondo al vector. Se la dimensione è inferiore alla capacità, l'elemento viene memorizzato e la dimensione è incrementata di uno. Se invece la dimensione è uguale alla capacità, non c'è posto per il nuovo elemento e il vector deve essere ridimensionato: ciò è abbastanza oneroso, infatti

1. viene allocato un nuovo array dinamico con una capacità superiore a quello precedente
2. il contenuto dell'array attuale viene copiato nel nuovo array
3. il vecchio array viene deallocato.

Per evitare di eseguire troppo spesso tale operazione, durante il ridimensionamento la capacità è aumentata di un numero cospicuo di nuovi elementi, anziché di un solo elemento alla volta. Ad esempio in molte implementazioni la capacità è raddoppiata ogni volta.

Nei vector si possono usare anche le seguenti operazioni “lente” (cioè che richiedono un tempo di esecuzione lineare nella dimensione di v, nel caso peggiore)

- v.resize(n), cambia la dimensione di v in n (operazione pesante, da svolgere raramente)
- v.insert(it,x), inserisce x nella posizione indicata dall'iteratore it
- v.erase(it), cancella l'elemento nella posizione indicata dall'iteratore it
- u=v, copia il vector v nel vector u.

Per usare insert e erase serve un iteratore, il cui impiego sarà spiegato più avanti in questo capitolo.

I vector sono molto utilizzati (e sono presenti, con altri nomi, in molti linguaggi di programmazione). E’ conveniente usare un vector al posto di un array tradizionale solo se la dimensione può variare nel tempo, mentre se la dimensione è fissa gli array sono comunque più efficienti.

E’ vivamente consigliato di passare ad una funzione i vector (al pari di tutte le altre strutture dati STL) per riferimento, in modo da evitare che il parametro sia una copia dell’argomento, anche nei casi in cui la funzione non deve modificare l’argomento.

Quando si passa un vector ad una funzione, non serve passare anche la dimensione, come invece avviene per gli array. Infatti la funzione può estrarre la dimensione usando il metodo `size`.

Ad esempio ecco una funzione che calcola la media degli elementi di un vector di double

```
double media(vector<double> &v) {
     int n=v.size();
     double somma=0;
     for(int i=0; i<n; ++i)
          somma=somma+v[i];
     return somma/n;
}
```

Si può comunque indicare che la funzione non deve modificare *v* usando una forma particolare di passaggio per riferimento: il passaggio per riferimento costante

```
double media(const vector<double> &v) { ...
```

In pratica, il passaggio per riferimento costante andrebbe usato in tutte quelle situazioni in cui si vuole avere la maggiore efficienza e il risparmio di occupazione di memoria del passaggio per riferimento, senza però consentire alla funzione di alterare, anche inavvertitamente, l’argomento passato nella chiamata.

Le liste

I vector consentono di gestire una collezione dinamica di elementi, potendo aggiungere ed eliminare elementi in fondo. Però sono poco adatti in alcune situazioni, dato che le operazioni di ricerca di un elemento e di inserimento e cancellazione in posizioni arbitrarie, possono essere svolte, nel caso peggiore, in tempo proporzionale al numero di elementi presenti.

Nelle situazioni in cui è necessario inserire e cancellare elementi in posizioni arbitrarie, possono essere utilizzate le liste. Infatti nelle liste tali operazioni sono eseguiti in un tempo costante, cioè indipendente dal numero di elementi.

Per usare le liste bisogna includere

```
#include <list>
```

Le liste si dichiarano indicando solo il tipo degli elementi tra parentesi angolari. Ad esempio

```
list<int> l1;

list<string> l2;
```

Ecco un elenco delle principali operazioni di list

- l.push_back(x), inserisce x in fondo a l
- l.push_front(x), inserisce x all'inizio di l
- l.pop_back(), rimuove l'ultimo elemento di l
- l.pop_front(), rimuove il primo elemento di l
- l.front(), restituisce il primo elemento di l
- l.back(), restituisce l'ultimo elemento di l
- l.clear(), svuota l
- l.size(), restituisce il numero di elementi di l
- l.empty(), restituisce true se l è vuota
- l.remove(x), elimina da l tutti i nodi che hanno chiave x

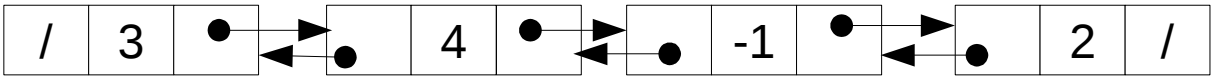
Come esempio di utilizzo di push_back, vediamo come si fa a leggere da tastiera una lista di numeri interi.

```
void leggi_lista(list<int> &l) {
     cout << “quanti elementi vuoi inserire ? “;
     int n; cin >> n;
     l.clear();
     for(int i=1; i<=n; ++i) {
          int k;
          cin >> k;
          l.push_back(k);
     }
```

```
}
```

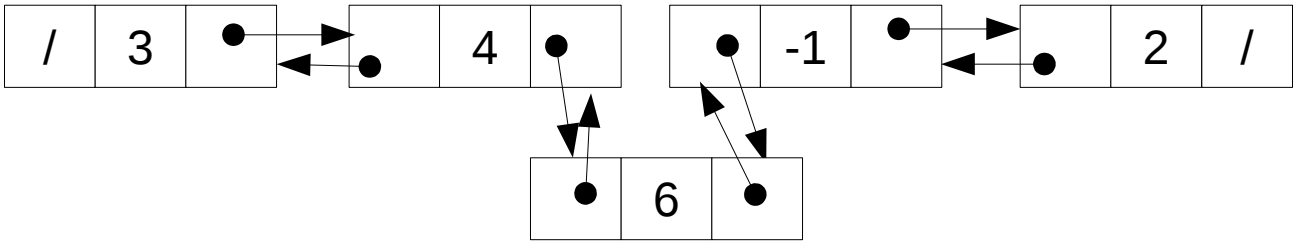
Le liste sono implementate in modo completamente diverso dai vector (e dagli array). Una lista contenente *n* elementi è costituita da *n* piccoli "nodi". Ogni nodo contiene il valore dell'elemento (detto *chiave*) e due puntatori (chiamati *prev* e *next*): prev contiene l'indirizzo del nodo precedente nella lista e next quello successivo. Sia il campo prev del primo elemento che il campo next dell'ultimo elemento contengono un valore speciale (ad esempio 0).

Perciò una lista di interi contenente i valori (3,4,-1,2) può essere rappresentata graficamente nel seguente modo



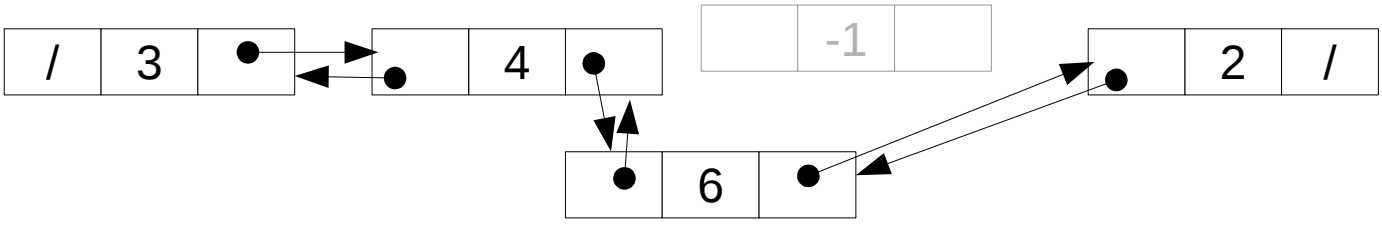
Ogni nodo è rappresentato con tre caselle, corrispondenti ai campi *prev*, *chiave* e *next*. Le frecce indicano i collegamenti, ad esempio il campo *next* del primo elemento contiene l'indirizzo del secondo elemento. Ogni nodo è allocato dinamicamente in maniera indipendente dagli altri nodi, per cui si può trovare in una posizione arbitraria in memoria centrale. Questa è la grande differenza tra array (o vector) e liste, infatti negli array gli elementi sono consecutivi in memoria e ciò può essere ottenuto solo allocando contemporaneamente tutti gli elementi. Nelle liste le relazione di "essere precedente" o "essere successivo" sono realizzate tramite i puntatori.

Per inserire un elemento in posizione arbitraria è sufficiente creare un nuovo nodo e collegarlo ai nodi esistenti. Ad esempio l'inserimento di 6 tra 4 e -1 produce la seguente lista



E' facile vedere che il numero di puntatori da cambiare è 4.

In modo analogo, è possibile eliminare velocemente un elemento. Ad esempio per togliere il -1 dalla lista



è sufficiente “scavalcarlo”, alterando due puntatori. L’elemento così potrebbe essere eliminato dalla memoria.

E’ abbastanza evidente che in una lista l’accesso mediante indice non sarebbe efficiente come negli array o nei vector, dato che gli elementi non sono consecutivi e per accedere all'i-esimo elemento serve passare per i precedenti *i-1* elementi. Pertanto tale operazione non è in generale disponibile (almeno con la sintassi delle parentesi quadre). Se si ha bisogno di una struttura dati in cui è frequente l’accesso mediante indice, allora è più conveniente usare un vector (o un array).

Iteratori

Un iteratore è un oggetto che serve a scorrere una collezione di dati o ad indicare una posizione all'interno di essa. Un iteratore quindi svolge le stesse funzioni di un indice per scorrere un array o un vector.

Per il momento, vediamo gli iteratori per una lista. Una variabile di questo tipo si dichiara

list< *tipo* >**::iterator** *nome*

in cui *tipo* è il tipo degli elementi della lista.

Gli iteratori consentono le seguenti 4 operazioni

- assegnamento (=)
- uguaglianza (==)
- accesso alla chiave (operatore *)
- passaggio al nodo successivo (operatore ++)

Per far partire un iteratore *it* dall'inizio di una lista *l* si usa `it=l.begin()`.

L'operatore ++ passa al nodo successivo e, quando *it* si trova già all'ultimo elemento, ++*it* passa ad un elemento fittizio che si trova “dopo l'ultimo”, indicato con `l.end()`.

Per cui, per scorrere una lista *l* con un iteratore *it* si usa il seguente schema

```
it=l.begin();
while( it != l.end() ) {
     // fai qualcosa con *it
     ++it;
}
```

o meglio, con un ciclo for

```
for(it=l.begin();  it!=l.end(); ++it) {
     //fai qualcosa con *it
}
```

Ad esempio per scrivere sullo schermo gli elementi di una lista *l*

```
void scrivi_lista(list<int> &l) {
     for(list<int>::iterator it=l.begin(); it!=l.end(); ++it)
          cout << *it << " ";
     cout << endl;
}
```

mentre per calcolare la media degli elementi

```
double media_lista(list<int> &l) {
     int somma=0;
     for(list<int>::iterator it=l.begin(); it!=l.end(); ++it)
          somma = somma + *it;
     return (double)somma/l.size();
}
```

La funzione *find* (presente nella libreria *algorithm*) cerca se esiste un elemento uguale a *x* all'interno di una lista l (o di altre collezioni) e fa uso degli iteratori. Infatti

- se *x* è presente in *l*, il risultato è un iteratore che ne "indica" la posizione;
- se *x* non c'è, il risultato è l.end()

La sua sintassi è

```
find(l.begin(), l.end(), x)
```

Tale operazione impiega un tempo proporzionale al numero di elementi presenti in *l*.

Un iteratore è usato per indicare un elemento nei seguenti metodi

- l.insert(it,x), inserisce un nuovo nodo con chiave x prima di it
- l.erase(it), cancella il nodo indicato da it.

Gli iteratori possono essere utilizzati anche nei vector, in modo analogo a quanto visto per le liste. Ovviamente, nei vector sono più scomodi da usare, rispetto agli indici e quindi se ne sconsiglia il loro impiego, tranne che per le operazioni di *insert* ed *erase*.

E' possibile anche avere iteratori che "vanno all'indietro" mediante l'operatore --.

Dalla versione C++11 è possibile usare un modo nettamente più semplice per scorrere una lista (o altre strutture dati STL, compresi i vector): il ciclo “for each”.

La sua sintassi per scorrere una lista *l* è

```
for(x : l) {
     //fai qualcosa con x
}
```

in cui *x* è una variabile dello stesso tipo degli elementi di x, che può essere direttamente dichiarata internamente al ciclo for.

Ad esempio per scrivere sullo schermo gli elementi di una lista *l*

```
void scrivi_lista(list<int> &l) {
     for(int x : l)
          cout << x << “ “;
     cout << endl;
}
```

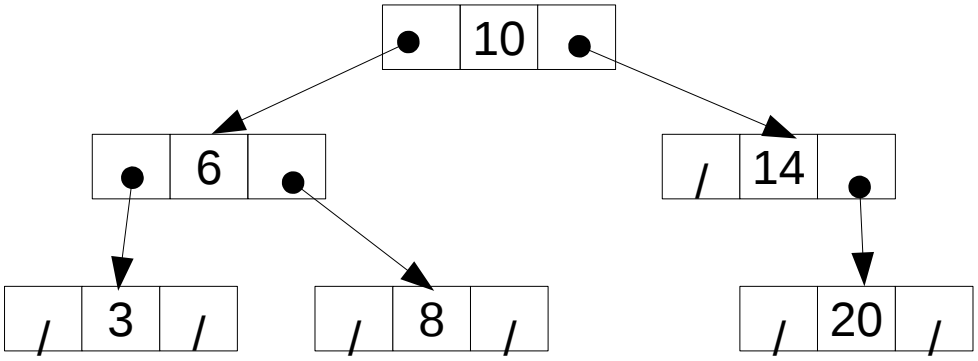
mentre per calcolare la media degli elementi

```
double media_lista(list<int> &l) {
     int somma=0;
     for(int x : l)
          somma = somma + x;
     return (double)somma/l.size();
}
```

Insiemi (Set)

Un insieme (set) è una struttura dati che consente di svolgere velocemente le operazioni di ricerca, inserimento e cancellazione. Nell'implementazione standard di set (che usa gli alberi binari di ricerca) queste operazioni sono eseguite in tempo O(*log* N), ove N è la cardinalità dell'insieme.

Ad esempio un albero binario di ricerca che rappresenta l’insieme {3,6,8,10,14,20} è ottenuto con il seguente schema



In modo simile ad una lista, un albero binario è costituito da nodi, ognuno dei quali contiene una chiave e due puntatori, chiamati rispettivamente *left* e *right*. La struttura di un albero non è quindi in generale lineare come nelle liste, ma “bidimensionale”.

Ogni nodo è allocato dinamicamente in modo indipendente dagli altri. In un albero binario di ricerca, ogni chiave è minore o uguale di tutti gli elementi che si trovano “verso destra” e maggiore o uguale delle chiavi che si trovano “verso sinistra”. Sfruttando tale proprietà, si riesce a cercare un elemento effettuando un unico percorso che parte dall’elemento più in alto (detto *radice*) e scendendo verso il basso. Se l’albero è “ben bilanciato”, la lunghezza di ogni percorso è minore o uguale al logaritmo in base 2 del numero degli elementi.

La caratteristica principale di un insieme è che non conta l'ordine di inserimento e non è possibile avere duplicati, ovvero elementi presenti più volte. I set hanno perciò i seguenti metodi:

- s.clear(), cancella l'insieme
- s.size(), restituisce il numero di elementi (cardinalità)
- s.find(x), restituisce un iteratore all'elemento che contiene x (oppure s.end() se x non è presente)
- s.insert(x), inserisce x in s (se x non è presente in s)
- s.erase(x), elimina x da s (se x è presente in s)

in cui s è una variabile di tipo set<T>, ove T è il tipo degli elementi.

Si noti che in un insieme non è possibile inserire un elemento in una posizione arbitraria: il posto in cui inserire un nuovo elemento è stabilito dall'implementazione dell'insieme.

L’inserimento e la cancellazione sono svolti attraverso la tecnica di gestione degli *alberi red-black*, che permette di svolgere tali operazioni in tempo logaritmico nel numero degli elementi, mantenendo l'albero “ben bilanciato”.

Per scorrere un insieme si usano, come per le liste, gli iteratori.

Ad esempio per scrivere un insieme di interi sullo schermo

```
void scrivi_insieme(set<int> &s) {
     for(set<int>::iterator it=s.begin(); it!=s.end(); ++it)
          cout << *it << “ “;
     cout << endl;
}
```

Gli elementi sono restituiti in ordine crescente.

In alternativa, si può usare il for each

```
void scrivi_insieme(set<int> &s) {
     for(int x : s)
          cout << x << " ";
     cout << endl;
}
```

Mappe (map)

Le mappe (map) sono una generalizzazione del concetto di array. L'array può essere visto come un'applicazione che ad ogni indice nell'insieme X={0,1,...,*n*-1} associa un valore appartenente ad un insieme Y (ad esempio numeri interi, reali, stringhe, ecc.).

Una mappa è invece è un'applicazione che ad ogni **chiave,** appartenente ad un generico insieme finito X, associa un **valore** appartenente ad un insieme Y.

La differenza di utilizzo tra map e array (o vector) è che al posto di un indice (che denota una posizione) si usa una chiave. Dal punto di vista implementativo le mappe sono realizzate in modo simile a quanto visto per i set, ovvero con alberi binari di ricerca. Ogni elemento della mappa è però visto come una coppia (chiave,valore).

I principali metodi e operazioni per le mappe sono

- m.find(k), cerca se esiste un elemento di chiave k, restituendo un iteratore a tale elemento (se esiste) oppure m.end() (se non esiste)
- m[k], restituisce il valore associato alla chiave k, oppure il valore di default (ad esempio 0 per i numeri) se la chiave k non compare in m
- m[k]=x, assegna il valore x alla chiave k
- m.clear(), svuota m
- m.erase(k), elimina da m la chiave k e il suo valore
- m.erase(it), elimina da m la coppia (chiave,valore) indicata dall' iteratore *it*

in cui m è una variabile di tipo map<T1,T2>, ove T1 è il tipo delle chiavi e T2 il tipo dei valori.

Le mappe sono molto utilizzate nei linguaggi di programmazione moderna (seppur chiamate con altri nomi, ad esempio dizionari). Uno degli utilizzi più interessanti è quello di calcolare una distribuzione di frequenza per una variabile discreta.

Ad esempio vediamo un programma che legge da tastiera una sequenza di parole possibilmente ripetute e per ogni parola distinta scrive il numero di volte che essa compare nella sequenza.

```
#include <map>
#include <iostream>
```

```
using namespace std;

int main() {
     map<string,int> freq;
     string s;
     cout << “inserisci il testo, digita fine per terminare\n”;
     while(true) {
          cin >> s;
          if(s==”fine”) break;
          freq[s]=freq[s]+1;
     }
     map<string,int>::iterator it;
     for(it=freq.begin(); it!=freq.end(); ++it) {
          cout << “parola: “ << it->first
               << “ frequenza: “ << it->second << endl;
     }
}
```

Si noti che l'iteratore su una mappa ha due operazioni di accesso (al posto di *)

- it->first, che restituisce la chiave dell'elemento corrente
- it->second, che restituisce il valore dell'elemento corrente.

L'operazione `freq[s]=freq[s]+1` ha due possibile comportamenti

- se s non è presente come chiave in freq (ovvero è la prima volta che s compare), allora freq[s] prima dell'operazione vale 0 e quindi freq[s] diventa 1
- se s è gia presente, allora freq[s] viene incrementato di 1.

Anche con le mappe è possibile usare il ciclo for each. L’unica differenza è che la variabile con cui scorrere una mappa è di tipo pair<K,V>, con i campi first e second. Per evitare di scrivere esplicitamente il tipo, si può usare **auto** come nell’esempio sottostante

```
     for(auto &p : freq) {
          cout << “parola: “ << p.first
               << “ frequenza: “ << p.second << endl;
     }
```

Esercizi

1. Scrivere una funzione che dati due vector u e v di numeri interi li concatena, memorizzando il risultato in un terzo vector w (anch'esso passato per riferimento).
2. Scrivere una funzione che dato un numero naturale n restituisce in un vector di numeri interi (passato per riferimento) tutti i fattori primi di n (ripetuti o non).

3. Scrivere una funzione che data una lista l1 di numeri interi, ricopia in una seconda lista l2 (passata per riferimento) gli elementi di l1 in ordine inverso.
4. Scrivere una funzione che data una lista l di numeri interi (passata per riferimento), toglie il primo elemento e lo mette in fondo.
5. Scrivere una funzione che dato un numero naturale n, restituisce in vector passato per riferimento l'elenco dei divisori propri di n.
6. Scrivere una funzione che dato un numero naturale n, restituisce in vector passato per riferimento le cifre binarie di n. Ad esempio se n=19, il vector deve contenere (1,0,0,1,1).
7. Scrivere una funzione che dato un numero naturale n, crea una matrice identità "dinamica" definita mediante un vector di vector di numeri reali (passato per riferimento).
8. Scrivere un programma che crea una mappa string/double contenente i tassi di cambio da alcune monete straniere in euro, legge da tastiera alcune cifre in denaro in valuta straniera (ognuna quindi con valore e moneta) e calcola la somma complessiva in euro.
9. Scrivere una funzione che dati due insiemi s1 e s2 di numeri interi calcola la loro intersezione (mettendo il risultato in un terzo insieme s3 passato per riferimento).

La gestione dei file

Introduzione

In C++ è possibile accedere ai dati contenuti nei file (memorizzati su unità di massa, come i dischi) attraverso il concetto di **flusso** (*stream*). Un flusso può essere associato ad un file ed è possibile leggere dati con l'operatore >> e scriverne con l'operatore <<.

Un flusso ad un file può essere aperto in varie modalità

- in (sola) lettura
- in (sola) scrittura
- in accodamento
- in lettura/scrittura.

L'apertura in sola lettura richiede che il file esista e che l'utente abbia i diritti per leggerlo, in caso contrario il flusso non viene aperto. Una volta aperto, si possono leggere i dati, ma non scriverci.

L'apertura in sola scrittura controlla se il file esiste:

- se non c'è, viene creato vuoto (si deve avere il diritto di creare un file nella directory corrispondente)
- se c'è già, viene svuotato (si deve avere il diritto di scrivere sul file).

Se l'operazione non è possibile, il flusso non è aperto. Se invece è possibile, si può iniziare a scrivere sul file (ovviamente non è possibile leggere, visto che in ogni caso il file è vuoto...).

Aprire un flusso in accodamento è simile ad aprirlo in scrittura, però se il file esiste, non viene svuotato, ma si inizierà a scrivere a partire dalla fine del file. Detto in altri termini, l'accodamento aggiunge nuovi dati in fondo ad un file, senza cancellare quelli già esistenti.

L'apertura in lettura/scrittura è la modalità più generale, ma anche quella più complicata da gestire e in queste dispense non sarà trattata.

Operazioni sui file

Per aprire un file si usa il metodo *open*, applicato ad una variabile di tipo *fstream*, che richiede due argomenti

- il pathname del file da aprire
- una costante che indica la modalità di apertura:
  - *ios::in* per l'apertura in lettura
  - *ios::out* per l'apertura in scrittura
  - *ios::out* | *ios::app* per l'apertura in accodamento

- *ios::in* | *ios::out* per l'apertura in lettura/scrittura

Ad esempio, per aprire un flusso in lettura sul file */home/marco/dati.txt*

```
fstream f;
f.open(“/home/marco/dati.txt”,ios::in);
```

Per controllare se il file è stato aperto correttamente, si può usare il metodo `f.is_open()`, che restituisce *true* in caso positivo, *false* altrimenti; oppure il metodo `f.fail()`, che restituisce *true* se il file non è stato aperto correttamente, *false* altrimenti.

Esistono vari modi di accedere ad un file, tra questi i più diffusi sono l'accesso **sequenziale** e l'accesso **casuale**.

L'accesso **sequenziale** è il meccanismo più semplice e più efficiente: si leggono i dati nello stesso ordine in cui sono presenti nel file e si scrivono i dati uno dopo l'altro. Tale modo di accesso assomiglia al modo con cui si accede usualmente ai dati di un array, ovvero uno dopo l'altro, in un ordine fissato.

L'accesso **casuale** consiste nel leggere o scrivere dati nel file in posizioni arbitrarie, che sono indicate tramite il metodo *seek*, in cui si deve specificare il numero esatto di byte (rispetto alla posizione attuale o all'inizio del file) da cui continuare a leggere o scrivere.

Poiché l'accesso casuale è difficile da gestire, tratteremo solo quello sequenziale. Gli operatori per leggere e scrivere su flusso sono quelli che si usano per leggere da tastiera e scrivere sullo schermo. In particolare, se *f* è un flusso aperto in lettura, allora

f >> variabile >> variabile >> ... >> variabile

legge dal flusso tanti valori separati da spazi (o tabulazioni, andate a capo) e li memorizza nelle variabili indicate.

Se si usa l'accesso sequenziale, non si possono “saltare” i dati contenuti in un file: ad esempio se si è interessati al quindicesimo dato, vanno comunque letti i primi 14.

Quando i dati in un file sono terminati, si ha una condizione chiamata “eof” (abbreviazione di *end of file).* E' possibile controllare se i dati sono terminati mediante il metodo `eof()`, che restituisce true se il file è finito, false altrimenti.

Se però i dati da leggere sono disposte su righe, allora per far sì che la condizione eof sia gestita correttamente è necessario leggere i dati con

f >> variabile >> variabile >> ... >> variabile >> ws

Infatti “ws” indica di saltare anche gli spazi (tabulazioni/andate a capo) presenti dopo l'ultimo dato letto. Se non si usasse *ws*, *eof*() non funzionerebbe correttamente: infatti dopo aver letto gli ultimi dati presenti nel file, il file non sarebbe ancora finito del tutto e quindi eof() sarebbe false: infatti nel file ci sarebbe ancora da leggere il carattere dell'andata a capo dell'ultima riga.

Il controllo eof() di solito si usa come condizione del ciclo while nel seguente schema

```
while(! f.eof() ) {
     //leggi i dati da f
     //fai qualcosa con i dati
}
```

Tale schema permette di trattare file con un numero di righe non noto a priori. Se invece tale informazione fosse nota al momento dell'esecuzione del programma, allora si potrebbe usare anche un ciclo for.

Per la scrittura su file si usa l'operatore <<. Quindi se f è un flusso aperto in scrittura, allora

f << *dato* << *dato* << ... << *dato*

scrive in modo sequenziale i dati sul file, esattamente nello stesso modo in cui sarebbero scritti sullo schermo. Inoltre si possono usare *endl* e *setw*, come già visto per cout.

Per concludere le operazioni su file, il flusso deve essere chiuso con il metodo `close()`. Tale operazione assume un'importanza maggiore se il file è aperto in scrittura (o in accodamento), infatti tutte le operazioni di scrittura su file sono salvate se e quando il file viene chiuso.

Alcuni esempi

Forniamo ora due esempi di gestione dei file. Nel primo esempio, scriviamo dei dati su file, disposti su due colonne.

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

int main() {
     fstream f;
     f.open(“dati.txt”,ios::out);
     if(! f.is_open() ) {
          cerr << “non riesco ad aprire il file\n”;
          exit(1);
     }
     for(int i=1; i<=100; i++) {
          int q=i*i:
          f << i << “ “ << q << endl;
     }
     f.close();
}
```

Nel secondo esempio, leggiamo i dati scritti dal programma precedente e ne calcoliamo le medie.

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

int main() {
     fstream f;
     f.open(“dati.txt”,ios::in);
     if(! f.is_open() ) {
          cerr << “non riesco ad aprire il file\n”;
          exit(1);
     }
     int somma_x=0, somma_y=0, num=0;
     while( ! f.eof() ) {
          int x,y;
          f >> x >> y >> ws;
          ++num;
          somma_x = somma_x + x;
          somma_y = somma_y + y;
     }
     f.close();
     double media_x=double(somma_x)/num,
            media_y=double(somma_y)/num;
     cout << “medie “ << media_x << “ “ << media_y << endl;
}
```

Nel terzo esempio scriviamo una funzione che riempe un vector con i dati della prima colonna del file `dati.txt`.

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

void riempi_vettore(vector<int> &xx) {
     fstream f;
     f.open(“dati.txt”,ios::in);
     if(! f.is_open() )
          return;
     while( ! f.eof() ) {
          int x,y;
```

```
            f >> x >> y >> ws;
            xx.push_back(x);
      }
      f.close();
}

int main() {
      vector<int> colonna;
      riempi_vettore(colonna);
      cout << “dati letti\n”;
      for(int i=0; i<colonna.size(); i++)
            cout << colonna[i] << endl;
}
```

Precisazioni sulla gestione dei permessi*

In questa sezione vedremo come sono gestiti i permessi sui file e sulle directory nel sistema operativo Linux. Meccanismi analoghi di protezione sono presenti anche negli altri sistemi operativi moderni.

I diritti gestiti da Linux sono 3, chiamati **r**, **w** e **x**. Per un file F, **r** è il diritto di aprire F in lettura, **w** è il diritto di aprire F in scrittura e **x** è il diritto di eseguire F (ovviamente ha senso solo se F è un programma). Per una directory D, **r** indica se è possibile ottenere il contenuto di D per sapere quali file e quali sottodirectory ci sono, **w** indica se è possibile modificare D creando, cambiando nome o cancellando un file (o una sottodirectory), infine **x** indica se è possibile attraversare D quando la si utilizza in un pathname.

Ogni file o directory (genericamente una risorsa R) ha un proprietario P (**owner**), che è di solito l’utente che ha creato R, ed un gruppo G di utenti, di cui fa parte P.

Il proprietario P può stabilire i diritti r, w e x su R per tre tipologie di utenti:

- se stesso
- i membri del gruppo G
- tutti gli altri utenti.

Con il comando `ls -l` è possibile vedere il proprietario e il gruppo di ciascuna risorsa e i rispettivi permessi.

Ad esempio nella riga

```
-rwxr-xr--  2 marco docenti        415 1 dic 2016 prova.cpp
```

si vede che per il file *prova.cpp* l’utente *marco* (il proprietario) ha tutti e tre i diritti (`rwx`), gli altri utenti del gruppo *docenti* hanno diritto di lettura ed esecuzione (`r-x`) ma non di scrittura. I rimanenti utenti possono solo leggerlo (`r--`).

Mediante il comando `chmod`, il proprietario (o l'amministratore) può stabilire quali diritti concedere o revocare alle tre tipologie di utenti su una determinata risorsa.

Mediante il comando `chown`, il proprietario (o l'amministratore) può assegnare la risorsa ad un altro proprietario o associarla ad un altro gruppo.

Esercizi

1. Scrivere un programma che scrive in un file il seno e il coseno (in due colonne) di tutti gli angoli compresi tra 0 e 360 gradi, con incremento di 5 gradi.
2. Scrivere un programma che scrive in un file l'elenco dei numeri primi da 2 a N (ove N è letto da tastiera).
3. Scrivere un programma che legge una sequenza di coppie x,y di numeri reali da un file e calcola i coefficienti della retta di regressione di y rispetto a x.
4. Scrivere un programma che legge i dati di un gruppo di studenti da file. Per ogni studenti sono memorizzati nome, cognome, sesso, altezza e peso. Il programma deve calcolare l'altezza e il peso medio, sia separatamente per maschi e per femmine, sia complessivamente.
5. Scrivere un programma che legge i dati dal file precedente e scrive in un secondo file solo i dati degli studenti con età superiore ai 22 anni.
6. Scrivere un programma che legge da un file una serie di prodotti, aventi una descrizione e un prezzo (ci sono al massimo 1000 prodotti) e li riscrive nello stesso file nello stesso ordine con i prezzi aumentati del 5%. Suggerimento: aprire il file in lettura, copiare i dati in un array di struct, chiudere il file, aprirlo in scrittura e scrivere i dati modificati.

La programmazione orientata agli oggetti

Introduzione

La programmazione orientata agli oggetti è uno dei concetti più importanti della programmazione moderna e praticamente tutti i nuovi linguaggi di programmazione sono orientati agli oggetti (C++, C#, Java, Scala, Python, ecc.).

L'idea di base della programmazione orientata agli oggetti è permettere al programmatore di definire nuovi tipi di dato, detti **classi**, che hanno una parte interna nascosta e una parte esterna visibile a tutto il resto del programma.

Una classe è costituita da due tipi principali di elementi:

- variabili, dette **campi** o **attributi,**
- funzioni, dette **metodi;**

inoltre può anche contenere altri tipi di elementi, ad esempio la definizione di tipi di dato.

Allo stesso modo con cui è possibile dichiarare variabili di tipo int, float, double, ecc., è possibile dichiarare variabili di una classe: la variabile conterrà un esemplare della classe che è chiamato **oggetto** (o in termini tecnici **istanza della classe**). Tutti gli oggetti di una classe avranno gli stessi campi e gli stessi metodi.

In particolare, i campi di un oggetto sono delle variabili e quindi due oggetti diversi della stessa classe possono avere valori diversi negli stessi campi.

Invece i metodi sono particolari funzioni che possono essere chiamate (o in termini tecnici **invocati**) “su un oggetto” della classe. Un metodo può accedere direttamente ai campi dell’oggetto in questione e può anche modificarne il valore; inoltre può anche avere degli argomenti con cui operare, che possono anche essere altri oggetti.

Ogni elemento della classe può essere essenzialmente **pubblico** o **privato**:

- un elemento è privato se è utilizzabile solo all'interno della classe;
- un elemento è pubblico se è utilizzabile da qualsiasi parte del programma.

Di solito i campi di una classe sono tutti privati, in modo che non sia possibile alterarne direttamente i valori, mentre i metodi possono essere pubblici o privati.

Se una classe ha solo campi privati (tale situazione è detta **incapsulamento**) allora i metodi pubblici sono l'unico modo di accesso ai dati: l'insieme di tali metodi agisce da **interfaccia** tra la classe e il resto del mondo. Detto in altri termini, l'incapsulamento permette di creare dati che possono essere manipolati in modo controllato: le uniche operazioni consentite sono solo quelle esprimibili attraverso

metodi pubblici.

Data una variabile *v* di una classe C, è possibile

- accedere ad un campo *f* di *v,* con la sintassi `v.f`
- chiamare (**invocare**) un metodo *m* di *v*, con la sintassi `v.m(`a_1`,...,`a_N`)`, ove a_1,...,a_N sono gli eventuali argomenti di *m*.

Ovviamente tali operazioni sono consentite o negate a seconda del contesto, seguendo la distinzione tra pubblico e privato.

Si noti che le struct sono un caso particolare di classi, in cui tutti gli elementi sono pubblici e solitamente sono campi.

Esempi illustrativi

In questa sezione vedremo due esempi di classi. Il primo esempio è una classe per gestire un conto corrente, il secondo esempio è una classe che gestisce i numeri complessi.

La classe conto corrente

Un conto corrente ha un titolare e un saldo corrente. Il conto corrente può essere aperto specificando il nome e cognome del titolare e un saldo iniziale. Si possono effettuare operazioni di versamento e di prelevamento, quest'ultima operazione è valida solo se il saldo consente il prelievo. Inoltre è possibile consultare il valore del saldo e verificare sullo schermo lo stato del conto corrente. Dall'esterno non è possibile accedere ai dati del conto corrente, se non tramite le operazioni previste. Sarebbe infatti insensato lasciare che si possa modificare il saldo direttamente senza svolgere una delle due operazioni di versamento o di prelievo.

Iniziamo con la definizione della classe

```
class conto_corrente {
private:
     string titolare;
     double saldo;
public:
     conto_corrente(string tit, double si);
     void versa(double importo);
     bool preleva(double importo);
```

```
     double dimmi_saldo();
     void scrivi();
};
```

La definizione della classe comprende

- il nome della classe, nell'esempio *conto_corrente*
- la parte privata, nell'esempio composta dai campi *titolare* e *saldo*
- la parte pubblica, che inizia con “public:”, composta in questo caso dai metodi *conto_corrente*, *versa*, *preleva*, *dimmi_saldo* e *scrivi*
- i metodi elencati nella classe sono normalmente solo prototipi, la loro definizione completa è effettuata successivamente.

La definizione di una classe può essere divisa in più sezioni, che sono delimitate dalle etichette “public:”, “private:” o “protected:”. Ogni elemento di una sezione è pubblico, privato o protetto (concetto che sarà spiegata nel prossimo capitolo), a seconda dell'etichetta della sezione. La prima sezione inizia con la parentesi graffa aperta ed è sempre privata (a meno di diversa indicazione), quindi private: nella prima sezione può essere omesso.

Il passaggio successivo è definire (**implementare**) i metodi.

```
conto_corrente::conto_corrente(string tit, double si) {
     titolare=tit;
     saldo=si;
}

void conto_corrente::versa(double importo) {
     if(importo>0)
          saldo=saldo+importo;
}

bool conto_corrente::preleva(double importo) {
     if(saldo<importo || importo<0) {
```

```
          cout << “impossibile prelevare\n”;
          return false; }
     saldo=saldo-importo;
     return true;
}

double conto_corrente::dimmi_saldo() {
     return saldo;
}

void conto_corrente::scrivi() {
     cout << “conto corrente di “ << titolare << endl;
     cout << “saldo attuale “ << saldo << endl;
}
```

Ogni metodo è implementato come se fosse una funzione, con due importanti differenze

- è indispensabile mettere prima del nome del metodo il nome della classe a cui appartiene secondo la sintassi *nome_della_classe* **::** *nome_del_metodo*
- il metodo può accedere direttamente ai campi dell'oggetto su cui è stato chiamato, ad esempio il metodo *versa* può usare il valore del campo *saldo* dell'oggetto su cui è stato invocato.

Alcuni metodi possono modificare l'oggetto su cui sono chiamati (ad esempio *conto_corrente* e *preleva*, *versa*), altri invece accedono ai dati, ma non li modificano (nell'esempio, *scrivi* e *dimmi_saldo*).

Il metodo “conto_corrente” è un metodo speciale, chiamato **costruttore**. Un costruttore si deve necessariamente chiamare come la classe, non ha risultato e quindi non si mette nemmeno void come tipo del risultato. Il costruttore non è mai invocato direttamente, ma viene chiamato automaticamente ogni volta che una variabile della classe viene dichiarata (ovvero quando viene creato un oggetto della classe). In pratica il costruttore serve ad inizializzare un oggetto, attribuendo un valore iniziale ai campi. Una classe può avere più costruttori e, in generale, può avere più metodi con lo stesso nome, a patto che differiscano nei parametri, nel numero o nel tipo (**overloading)**.

Vediamo ora un main che crea e utilizza oggetti della classe.

```
int main() {
```

```
    conto_corrente c(“Mario Rossi”,1000);
    c.versa(500);
    c.preleva(200);
    cout << c.dimmi_saldo() << endl;
    c.versa(300);
    c.scrivi();
}
```

All’inizio viene creato un oggetto della classe, tramite la variabile *c*. Gli oggetti sono inizializzati mediante il costruttore, la sintassi per dichiarare una variabile è infatti:
nome_classe nome_variabile **(** *argomenti_costruttore* **)**

Poi sono invocati vari metodi.

Il programma scriverà sullo schermo 1300 e successivamente

```
conto corrente di Mario Rossi
saldo attuale 1600
```

Si noti che i campi privati (*saldo* e *titolare*) sono inaccessibili dall'esterno e quindi ogni tentativo di accedervi direttamente, ad esempio con

```
double s=c.saldo;
```

oppure con

```
c.saldo=100000;
```

sono vietati e causano errori in compilazione.

Su concessione del programmatore, è comunque possibile accedere indirettamente al valore del saldo tramite il metodo *dimmi_saldo*. Tale accesso consente solo di avere il saldo corrente, ma non è permesso modificarlo. Infatti un'invocazione del tipo

```
c.dimmi_saldo()=100000;
```

provoca un errore in compilazione, analogamente a quanto farebbe `sqrt(x)=3;`

E' importante capire che esiste una differenza consistente tra un campo pubblico e un campo privato, come *saldo*, a cui si può accedere tramite il metodo pubblico *dimmi_saldo*.

Nel primo caso, ogni accesso esterno sarebbe possibile e quindi sarebbe consentito cambiare in modo arbitrario il valore di detto campo.

Nel secondo caso, il programmatore consente di conoscere il valore del campo, ma non di poterlo cambiare dall'esterno.

E' anche possibile immaginare una situazione in cui un campo privato *c* ammette sia un metodo per

avere il valore di *c*, sia un metodo per cambiare (in modo controllato) il valore di *c*. Nuovamente, questa situazione è comunque diversa da definire *c* come campo pubblico: infatti nel primo caso, l'accesso a *c* avviene sempre e solo tramite i metodi previsti dal programmatore.

La classe complesso

In questo secondo esempio vediamo una classe che consente di creare e operare con numeri complessi.

```
class complesso {
     double re, im;
public:
     complesso();
     complesso(double x,double y);
     void scrivi();
     double modulo();
     complesso somma(const complesso &c);
};
```

La parte privata è composta dai campi *re* e *im*, di tipo *double*, mentre la parte pubblica è composta dal costruttore (in due versioni distinte) e dai metodi "*scrivi*", "*modulo*" e "*somma*".

L'implementazione dei metodi è

```
complesso::complesso() {
     re=0;
     im=0;
}

complesso::complesso(double x,double y) {
     re=x;
     im=y;
}
```

```
void complesso::scrivi() {
     cout << re << “+i” << im;
}

double complesso::modulo() {
     double m=sqrt(re*re+im*im);
     return m;
}

complesso complesso::somma(const complesso &c) {
     complesso ris;
     ris.re=re+c.re;
     ris.im=im+c.im;
     return ris;
}
```

Infine, un possibile *main* che utilizza la classe è

```
int main() {
     complesso c1(2,3);
     c1.scrivi(); cout << endl;
     double m=c1.modulo();
     cout << “ modulo=” << m << endl;
     complesso c2(1,2);
     complesso c3;
     c3=c1.somma(c2);
     c3.scrivi(); cout << endl;
}
```

Sono creati due oggetti: *c1*, il cui valore iniziale è *2+3i*, e *c2*, il cui valore iniziale è *1+2i*.

Poi sono invocati i vari metodi. Si noti che il metodo *somma* ha come parametro un altro oggetto della classe: il suo effetto è quello di addizionare due numeri complessi, memorizzando il risultato in un nuovo oggetto.

Si noti che, per motivi di efficienza, *c2* è passato per riferimento a costante, senza che sia necessario crearne una copia. In generale conviene sempre passare gli oggetti per riferimento (o tramite puntatore), tranne casi particolari. Se l'oggetto passato come parametro non viene modificato dal metodo, come in questo caso, allora ha senso passarlo per riferimento a costante.

Il primo costruttore di complesso si chiama **costruttore per default** in quanto non ha parametri. Esso serve a dichiarare variabili di una classe senza indicare altro (cioè senza gli argomenti del costruttore e senza nemmeno le parentesi tonde), come avviene nella prima riga del metodo "somma". Nell'esempio, il costruttore di default della classe *complesso* crea un numero complesso nullo.

Metodi inline

E' possibile definire un metodo all'interno del costrutto class. In tal caso il metodo è considerato e compilato come metodo "inline", ovvero ogni volta che il metodo è invocato, il compilatore sostituisce la chiamata del metodo con le istruzioni (opportunamente adattate) del metodo stesso.

L'uso dei metodi inline, se da un lato produce programmi leggermente più efficienti, dall'altro produce file eseguibili più lunghi. Quindi tale utilizzo va limitato solo a metodi corti, aventi poche istruzioni.

Ad esempio modifichiamo la classe *conto_corrente* definendo come metodi inline sia il costruttore, sia il metodo *dimmi_saldo*.

```
class conto_corrente {
     string titolare;
     double saldo;
public:
     conto_corrente(string tit, double si) {
          titolare=tit;
          saldo=si; }
     void versa(double importo);
     bool preleva(double importo);
     double dimmi_saldo() {
```

```
          return saldo; }
     void scrivi();
};
```

Ovviamente i due metodi inline sono già stati implementati internamente e quindi non devono essere definiti al di fuori di *class*.

Elementi statici

E' possibile dichiarare che un elemento della classe è **statico**, ovvero si riferisce all'intera classe e non al singolo oggetto. Ad esempio è possibile dichiarare un metodo *somma* che addiziona due numeri complessi restituendo come risultato un terzo numero complesso, in modo che entrambi gli operandi siano argomenti del metodo.

```
class complesso {
     double re, im;
public:
     complesso();
     complesso(double x,double y);
     void scrivi();
     double modulo();
     static complesso somma(const complesso &c1,const complesso &c2);
};
```

L’implementazione del metodo *somma* è

```
complesso complesso::somma(const complesso &c1,const complesso &c2) {
     complesso r;
     r.re=c1.re+c2.re;
     r.im=c1.im+c2.im;
     return r;
}
```

Il main che usa il metodo somma è

```
int main() {
     complesso c1(2,3), c2(1,2);
     complesso c3;
     c3=complesso::somma(c1,c2);
     c3.scrivi(); cout << endl;
}
```

Si noti che il metodo, essendo statico, è chiamato sulla classe “complesso” e non sul singolo oggetto. Inoltre tale metodo non altera né *c1* né *c2*. Quindi un metodo statico si usa quando si vuole definire un’operazione tra oggetti della classe, senza che sia invocata su uno degli argomenti.

Un altro possibile impiego dei metodi statici è per creare oggetti della classe (tali metodi sono chiamati **factory**).

Un metodo statico assomiglia un po’ ad una funzione normale, ma esiste un’importante differenza: il primo, essendo un metodo della classe, può accedere alla parte privata dei due numeri complessi, mentre la seconda non può farlo.

I campi statici sono campi i cui valori sono comuni a tutti gli oggetti della stessa classe: un campo statico corrisponde ad una variabile della classe (anziché del singolo oggetto).

Gestione dinamica degli oggetti

Un modo molto diffuso e semplice di gestire gli oggetti è la gestione dinamica tramite puntatori.

Il programmatore può creare un oggetto con *new* e distruggerlo con *delete*.

Ad esempio

```
int main() {
     complesso *c1;
     c1=new complesso(2,3);
     c1->scrivi(); cout << endl;
     double m=c1->modulo();
     cout << “ modulo=” << m << endl;
     complesso *c2=new complesso(1,2);
     complesso c3=c1->somma(*c2);
```

```
    c1->scrivi(); cout << “+”;
    c2->scrivi(); cout << “=”;
    c3.scrivi();  cout << endl;
    delete c1;
    delete c2;
}
```

Per creare un oggetto in modo dinamico si deve usare la sintassi

new *nome_classe* **(** argomenti_costruttore **)**

mentre per eliminarlo è sufficiente

delete *puntatore*

Per accedere a campi e metodi di un oggetto gestito tramite puntatori si usa la freccia `->` al posto del punto, allo stesso modo in cui si usa con un puntatore ad una struct.

Si noti che con questo sistema è possibile creare oggetti in una parte del programma e usarli in un'altra parte, ad esempio si può definire una funzione che crea un oggetto e restituisce il suo indirizzo come risultato

```
complesso* leggi_da_tastiera() {
    double r,i;
    cin >> r >> i;
    complesso *p=new complesso(r,i);
    return p;
}
```

La gestione tramite puntatori è molto utilizzata e, in molti linguaggi recenti (Java e C#, ad esempio), è l'unico modo di gestire un oggetto (seppur con una forma depotenziata di puntatore).

L'uso dei puntatori e degli oggetti dinamici ha comunque dei risvolti negativi. Innanzitutto è indispensabile eliminare un oggetto quando non serve più: se ci si dimentica di farlo, l'oggetto rimane in memoria fino alla fine dell'esecuzione del programma. Per eliminare un oggetto è necessario avere un puntatore ad esso: se per qualche motivo l'indirizzo dell'oggetto non è più conservato in qualche puntatore, l'oggetto (pur essendo inutilizzabile) non è cancellabile.

Ad esempio in

```
void prova() {
    complesso *p=new complesso(3,2);
```

```
    ...
    // mi dimentico di eseguire delete p
}
```

l'indirizzo dell'oggetto era memorizzato in *p*, ma dopo che è terminata la funzione, dell'oggetto se ne è persa ogni traccia e non può essere più eliminato.

Un altro problema è dovuto al fatto che si può copiare un indirizzo di un oggetto su più puntatori. Ad esempio

```
complesso *p1, *p2;
p1=new complesso(5,2);
p2=p1;
```

ora sia *p1* che *p2* puntano allo stesso oggetto. Supponiamo di eliminare l'oggetto con l'istruzione `delete p1`, il puntatore p2 continuerà a contenere il vecchio indirizzo dell'oggetto: ogni tentativo di accedervi è potenzialmente un errore e potrebbe mandare in “crash” il programma.

Classi e compilazione separata

E’ possibile utilizzare le classi in un programma suddiviso in più file sorgenti. La prassi comunemente seguita è quella di creare due file per ogni classe utilizzata:

- un file intestazione (con estensione .h), contenente la definizione della classe
- un file sorgente (con estensione .cpp), contenente l’implementazione dei metodi

Ad esempio il file *complesso.h* potrebbe contenere

```
class complesso {
    double re, im;
public:
    complesso() {
        re=0;
        im=0; }

    complesso(double x,double y) {
        re=x;
```

```
          im=y; }

     void scrivi();
     double modulo();
     complesso somma(complesso &c2);
};
```

mentre il file *complesso.cpp* potrebbe contenere

```
complesso complesso::somma(complesso &c2) {
     complesso r;
     r.re=re+c2.re;
     r.im=im+c2.im;
     return r;
}

void complesso::scrivi() {
     cout << re << "+i" << im;
}

double complesso::modulo() {
     double m=sqrt(re*re+im*im);
     return m;
}
```

Per usare la classe complesso in un altro file sorgente è quindi sufficiente includere il file *complesso.h*.

Ovviamente, in fase di compilazione dovrà essere creato il file oggetto *complesso.o* e collegato a tutto il resto tramite il linker.

Esercizi

1. Definire una classe *orologio* che ha i seguenti metodi pubblici
   a) costruttore per default, crea un orologio che segna la mezzanotte
   b) void avanza(), fa avanzare l'orologio di un minuto
   c) void visualizza(), visualizza l'orario corrente nella forma HH:MM
2. Definire una classe *frazione*, che consente di lavorare con le usuali frazioni, prevedendo le quattro operazioni aritmetiche e mantenendo le frazioni sempre ridotte ai minimi termini (implementare quindi un metodo privato che semplifica una frazione)
3. Definire una classe *veicolo,* che gestisce la posizione e i movimenti di un veicolo a motore ideale posizionato in un piano cartesiano e che ha i seguenti metodi pubblici
   a) costruttore, crea un veicolo posizionato nell'origine, a motore spento e direzionato lungo l'asse delle x (verso delle x crescenti)
   b) void giraOrario(), ruota il veicolo di 90 gradi in senso orario
   c) void giraAntiOrario(), ruota il veicolo di 90 gradi in senso antiorario
   d) void accendi(), accende il motore
   e) void spegni(), spegne il motore
   f) void avanza(n), simula il trascorrere di n unità di tempo, se il motore è acceso il veicolo si sposta a velocità unitaria lungo il piano nella direzione in cui è orientato
   g) void posizione(), visualizza la posizione del veicolo e lo stato del motore
4. Definire una classe *distributore_di_bevande* avente i seguenti metodi pubblici
   a) costruttore, crea un distributore di bevande, con 0 euro in cassa
   b) void inserisci_moneta(int *c*), inserisce una moneta da *c* centesimi di euro, controllando se *c* è un importo di una moneta esistente
   c) void annulla(), restituisce i soldi inseriti dall'utente
   d) void seleziona(*b*), l'utente seleziona la bevanda *b*, la macchina controlla che abbia inserito una cifra sufficiente per quella bevanda. In caso affermativo scrive sullo schermo che la bevanda è consegnata e quant'è il resto da restituire, inoltre incamera la somma nella cassa. In caso contrario segnala “cifra insufficiente” all'utente.
   e) double cassa(), restituisce come risultato il totale in euro presente in cassa

   Prevedere un certo numero di bevande, ognuna con un proprio prezzo.
5. Definire una classe *matrice2* che permette di gestire matrici 2x2 e aventi i seguenti metodi pubblici

a) costruttore con quattro parametri (a,b,c,d), crea una matrice $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$

b) costruttore per default, crea una matrice nulla

c) void scrivi(), scrive sullo schermo la matrice

d) matrice2 somma(const matrice2 &m), restituisce come risultato la somma della matrice con la matrice m

e) matrice2 prodotto(const matrice2 &m), idem per il prodotto

f) double det(), restituisce il determinante

g) matrice2 inversa(), restituisce la matrice inversa, se il determinante è diverso da 0, altrimenti restituisce la matrice nulla

h) void trasponi(), cambia la matrice trasponendola

6. Definire una classe *filetto* che gestisce una partita a filetto (detto anche tris) avente i seguenti metodi pubblici

   a) costruttore, crea uno schema 3x3 vuoto

   b) void crocetta(x,y), pone una crocetta nella cella (x,y), scrive un messaggio di errore se la cella è già occupata o non tocca a lui (a lui spetta il primo turno, il terzo, ecc.)

   c) void pallino(x,y), pone un pallino nella cella(x,y), scrive un messaggio di errore se la cella è già occupata o non tocca a lui (a lui spetta il secondo turno, il quarto, ecc.)

   d) void situazione(), scrive sullo schermo lo schema del gioco

   e) int controllo(), restituisce 0 se nessuno ha ancora vinto, 1 se ha vinto il giocatore con la crocetta, 2 se ha vinto quello con il pallino

7. Scrivere una classe *memoria* avente i seguenti metodi pubblici

   a) costruttore, crea una memoria inizialmente vuota

   b) void inserisci(str), inserisce la stringa str nella memoria

   c) void visualizza(), scrive sullo schermo le stringhe presenti in memoria nell'ordine in cui sono state inserite

   d) void visualizzaR(), scrive sullo schermo le stringhe presenti in memoria nell'ordine inverso in cui sono state inserite

   e) int conta(c), restituisce come risultato il numero di stringhe che iniziano con il carattere c

   f) void azzera(), svuota la memoria

Memorizzare le stringhe in un'opportuna struttura dati interna alla classe (potrebbe essere un array, un vector, un lista,...)

8. Definire una classe *studente* avente i seguenti metodi pubblici
    a) costruttore(nome,cognome,corso_di_laurea), che crea un nuovo studente a inizio carriera
    b) void sostieni_esame(materia,voto), aggiorna i dati dello studente con un nuovo esame
    c) double media(), restituisce la media dei voti, 0 se non ne ha sostenuti
    d) void certificato(), scrive sullo schermo i dati dello studente e l'elenco degli esami con i rispettivi voti

    Memorizzare le materie e i voti in un'opportuna struttura dati interna alla classe (potrebbe essere un array, un vector, un lista,...)
9. Definire una classe *rubrica_telefonica* avente i seguenti metodi pubblici
    a) costruttore, crea una rubrica inizialmente vuota
    b) void inserisci(nome, cognome, telefono), inserisce una nuova voce nella rubrica, anche telefono è di tipo stringa
    c) string cerca(nome,cognome), cerca il telefono di una persona in base a nome e cognome, se non lo trova restituisce la stringa “persona inesistente”
    d) int cognomi(c), restituisce il numero di persone che hanno il cognome che inizia con il carattere c
    e) void elimina(nome,cognome), elimina l'elemento indicato, scrivendo un messaggio sullo schermo se l'elemento non è presente

    Memorizzare i dati delle persone presenti nella rubrica in opportune strutture dati interne alla classe. Se si vuol semplificare l'esercizio accorpare cognome e nome insieme, in un'unica stringa.
10. Definire una classe *biblioteca* avente i seguenti metodi pubblici
    a) costruttore, crea inizialmente una biblioteca vuota
    b) void leggi(nomefile), apre in lettura il file contenente i dati in alcuni libri (ad esempio titolo e autore) e inserisce nella biblioteca i libri in questione. Un problema tecnico si ha sia con il titolo che con l'autore, dato che potrebbe contenere degli spazi e la lettura con >> di una stringa non consente la presenza di spazi. Quindi o si eliminano gli spazi usando il carattere “_” oppure si legge da file tramite il metodo getline, mettendo titolo in una riga e autore in un'altra
    c) bool presente(titolo), cerca un libro in base al titolo, restituisce true se c'è, false altrimenti
    d) int libri_scritti(autore), restituisce il numero di libri scritti dall'autore
    e) void acquista(titolo,autore), inserisce un nuovo libro
    f) void salva(nomefile), salva il contenuto nel file, usando lo stesso “formato” previsto per il

metodo leggi

Memorizzare i dati dei libri in opportune strutture dati interne alla classe. Per semplificare un po' l'esercizio, si usino solo libri con un solo autore, altrimenti per gestire gli autori di un libro occorrerebbe un'ulteriore struttura dati.

Aspetti avanzati della programmazione orientata agli oggetti

Vedremo in questo capitolo alcuni aspetti avanzati della programmazione orientata agli oggetti.

Array e collezioni di oggetti

Per definire un array di oggetti è indispensabile che la classe abbia un costruttore per default. Infatti se si dichiara, ad esempio, un array di 10 numeri complesso con

```
complesso arr[10];
```

il compilatore invoca 10 volte il costruttore per default della classe complesso per creare gli elementi arr[0], …, arr[9]. L'assenza del costruttore per default provoca un errore in compilazione.

Nel caso in cui sia necessario creare un array di oggetti, per inizializzare ognuno di essi separatamente, ci sono due strade.

La prima è quella di avere un costruttore di default e un metodo che successivamente inizializza l'oggetto attraverso dei parametri, da usare un po' come se fosse il “vero” costruttore.

In alternativa è possibile usare un array di puntatori ad oggetti, in modo che ognuno possa essere inizializzato al momento della new

```
conto_corrente *p[10];
p[0]=new conto_corrente(“Mario Rossi”,1000);
p[1]=new conto_corrente(“Luisa Neri”,2000);
...
```

Allo stesso modo è possibile usare un vector di puntatori ad oggetti

```
vector<conto_corrente*> p;
p.push_back(new conto_corrente(“Mario Rossi”,1000));
p.push_back(new conto_corrente(“Luisa Neri”,2000));
...
```

Composizione

Si può definire una classe tramite **composizione**, cioè contenente al suo interno uno o più oggetti di altre classi, memorizzati in campi nella nuova classe.

Ad esempio, definiamo una classe *punto*, che rappresenta un punto in uno spazio bidimensionale)

```
class punto {
     double x,y;
public:
     punto(double xx,double yy) {
          x=xx;
          y=yy; }
     void scrivi() {
          cout << “(“ << p.x << “,” << p.y << “)”;
     }
     void trasla(double dx,double dy) {
          x=x+dx;
          y=y+dy;
     }
};
```

e una classe *rettangolo*, che rappresenta un rettangolo tramite due punti: il vertice in alto a sinistra e quello in basso a destra

```
class rettangolo {
     punto p1,p2;
     string colore;
public:
     rettangolo(double x1,double x2,double y1,double y2,string c)
          : p1(x1,y1), p2(x2,y2) {
               colore=c;
     }
     void scrivi();
     void trasla(double dx,double dy) {
          p1.trasla(dx,dy);
          p2.trasla(dx,dy);
```

```
    }
};
void rettangolo::scrivi() {
    cout << “rettangolo di colore “ << colore << “ di vertici “;
    p1.scrivi();
    cout << “ e “;
    p2.scrivi();
}
```

Il costruttore di *rettangolo* è molto particolare: esso invoca i costruttori dei due campi *p1* e *p2* di tipo *punto* passando i rispettivi argomenti. Queste invocazioni non si possono scrivere all'interno del corpo del costruttore poiché in C++ i costruttori non possono essere invocati esplicitamente, ma vanno indicate con una sintassi apposita

costruttore(*parametri_costruttore*) : *campo1*(*parametri1*), …., *campoN*(*parametriN*) {

istruzioni_costruttore

}

in cui *campo1*, …., *campoN* sono i campi da inizializzare chiamando i rispettivi costruttori.

Gli oggetti contenuti in un altro oggetto, come i punti contenuti in un rettangolo, sono ad uso esclusivo dell'oggetto che li contiene e ne condividono la vita: quando un rettangolo viene creato, sono creati i due vertici p1 e p2; quando il rettangolo viene deallocato, anche i due punti subiscono la stessa sorte.

E' però possibile avere una forma più debole di composizione, in cui al posto di avere degli oggetti direttamente dentro un altro oggetto, l'oggetto contenitore ha dei campi di tipo puntatore, in cui sono memorizzati gli indirizzi degli oggetti da contenere.

Ad esempio, una variante di rettangolo può essere

```
class rettangolo {
    punto *p1,*p2;
    string colore;
public:
    rettangolo(punto *q1,punto *q2,string c) {
        p1=q1;
        p2=q2;
```

```
        colore=c;
    }
    ...
};
```

In questo modo è possibile creare un rettangolo a partire da due punti già creati in precedenza, passando al costruttore i loro indirizzi. I punti hanno una vita propria e possono essere deallocati anche molto tempo dopo la deallocazione del rettangolo di cui facevano parte.

Distruttore

In alcune situazioni può essere utile definire un metodo **distruttore**, che viene invocato appena prima che un oggetto sia deallocato. In pratica, il distruttore è il contrario del costruttore: infatti mentre il secondo serve ad inizializzare l'oggetto, il primo si usa per compiere delle azioni di “chiusura” dell'oggetto.

Il distruttore di una classe ha un nome che è formato dal carattere tilde ~ e dal nome della classe e non ha parametri.

Una situazione in cui ha senso definire un distruttore è quando una classe gestisce un oggetto interno tramite un puntatore: nel costruttore l'oggetto interno è allocato tramite *new*, mentre nel distruttore l'oggetto è deallocato con *delete*.

Ad esempio

```
class cerchio {
    punto *centro;
    double raggio;
public:
    cerchio(double x,double y,double r) {
        centro=new punto(x,y);
        raggio=r;
    }
    ~cerchio() {
        delete punto;
    }
```

```
    void trasla(double dx,double dy) {
        centro->trasla(dx,dy);
    }
};
```

Ereditarietà

Un altro importante concetto della programmazione orientata agli oggetti è l'**ereditarietà**. In modo per alcuni versi analogo alla composizione, l'ereditarietà consente di definire una nuova classe a partire da una classe già esistente.

La nuova classe, detta **classe derivata** (o classe **figlia**) eredita dalla classe già esistente, detta **classe base** (o classe **padre**), tutti i metodi e i campi in essa definita. Si dice anche la classe derivata **estende** la classe base. Inoltre la classe derivata può

- definire nuovi campi
- definire nuovi metodi
- implementare in modo diverso alcuni dei metodi ereditati.

Invece nella classe derivata non si può

- eliminare campi o metodi ereditati
- ridefinire i campi ereditati
- cambiare il prototipo dei metodi ereditati.

Forniamo ora come esempio due classi, in cui la seconda estende la prima.

```
class persona {
    string nome, cognome;
    double altezza, peso;
public:
    persona(string n,string c,double a,double p) {
        nome=n;
        cognome=c;
```

```
          altezza=a;
          peso=p; }
     void presentati();
     double indice_massa_corporea() {
          return peso/(altezza*altezza)
     }
};
void persona::presentati() {
     cout << “sono “ << nome << “ e “ << cognome << endl;
     cout << “sono alto “ << altezza << “ cm e peso “ << peso << “ kg\n”;
}

class studente : public persona {
     string corso_di_laurea;
     int numero_esami;
public:
     studente(string n,string c,double a,double p,string l)
          : persona(n,c,a,p) {
          corso_di_laurea=l;
          numero_esami=0; }
     void presentati();
     void sostieni_esame() {
           ++numero_esami;
     }
};
void studente::presentati() {
     persona::presentati();
     cout << “sono iscritto a “ << corso_di_laurea << “ e ho dato “ << numero_esami << “ esami\n”;
```

```
}
```

In questo esempio la classe “studente” eredita dalla classe persona i campi *nome*, *cognome*, *altezza* e *peso* e i metodi *presentati* (che però viene ridefinito) e *indice_massa_corporea*. Inoltre *studente* possiede propri campi (*corso_di_laurea* e *numero_esami*) e propri metodi (*sostieni_esame*).

Si noti che il costruttore di *studente* richiama il costruttore di *persona*. Infatti nella dichiarazione del costruttore

```
studente(string n,string c,double a,double p,string l)
```

la notazione

```
: persona(n,c,a,p)
```

indica proprio questo fatto.

Il metodo *presentati* di *persona* è stato implementato in modo differente nella classe *studente*. Tale situazione si chiama **overriding** (sovrapposizione) ed è perfettamente coerente con la filosofia della programmazione orientata agli oggetti.

In realtà l'implementazione di *presentati* di *studente* richiama la versione originaria nella sua prima linea di codice (`persona::presentati()`). In questo caso è indispensabile perché *studente* non può accedere alla parte privata di *persona*, anche se questi campi sono stati ereditati. Infatti tali campi sono privati e quindi inaccessibili dall'esterno della classe persona.

Per definire dei campi (e metodi) che siano accessibili alle classi figlie, ma non alle altre classi, è sufficiente dichiararli come **protetti** (dopo l'etichetta protected:).

Ad esempio

```
class persona {
...
protected:
     string num_telefono;
...
}

class studente : public persona {
// studente può accedere a num_telefono
}
```

L'ereditarietà si fonda sul principio che ogni oggetto della classe derivata può essere utilizzato come se fosse un oggetto della classe base. Ad esempio

```
int main() {
     persona p(“Mario”,”Rossi”,1.80,70),
          s(“Paola”,”Verdi”,1.70,55,”Fisica”);
     p.presentati();
     s.presentati();
     cout << p.indice_massa_corporea() << endl;
     cout << s.indice_massa_corporea() << endl;
}
```

Ovviamente, su *s* è possibile invocare il metodo *sostieni_esame*, mentre su *p* non è possibile.

In C++ una classe può essere contemporaneamente sottoclasse di più classi già esistenti.

Ad esempio

```
class studente_lavoratore : public studente, public lavoratore {
...
}
```

Tale caratteristica si chiama **ereditarietà multipla** ed è abbastanza rara nei linguaggi ad oggetti. Infatti, un problema che si pone è cosa fare quando esistono campi o metodi con lo stesso nome in due o più classi base, ad esempio *nome* e *cognome* sia in *studente* che in *lavoratore*. Pertanto in molti linguaggi moderni (Java, C#, Python, ecc.) è ammessa solo l'ereditarietà singola.

Polimorfismo e metodi virtuali

In C++ è possibile memorizzare uno *studente* in una variabile di tipo *persona*. L'effetto è però che lo studente è “retrocesso” ad essere una persona

```
int main() {
     studente s(“Luca”,”Neri”,1.81,72,”Matematica”);
     persona p; // definire un costruttore di default in persona !
     p=s;
     p.presentati();
```

```
}
```

La variabile *p*, pur contenendo una copia di uno *studente*, considera il proprio contenuto come un oggetto di tipo *persona*. Quindi `p.presentati()` non scriverà il corso di laurea, né sarà possibile invocare su *p* il metodo *sostieni_esame*.

Ciò accade perché è il compilatore a decidere se un metodo può essere invocato e, in caso affermativo, quale versione eseguire (quella di *persona* o quella di *studente* ?) sulla base del tipo della variabile: *p* è di tipo *persona*, quindi si usano i metodi di quella classe.

Ovviamente, un *conto_corrente* non può essere memorizzato in una variabile di tipo *persona* o di tipo *studente*, perché tra queste classi non vi sono relazioni di sottoclasse. E nemmeno una *persona* può essere memorizzata in uno *studente* (si pensi ad esempio che non avrebbe senso il corso di laurea).

Il comportamento cambia se il metodo *presentati* è dichiarato come metodo virtuale e gli oggetti sono gestiti mediante puntatori.

Quindi, facendo tali modifiche

```
class persona {
     ...
     virtual void presentati();
     ...
};
...
class studente : public persona {
     ...
     virtual void presentati();
};
...
int main() {
     persona *p=new persona(“Carla”,”Rossi”,1.70,56);
     persona *s=new studente(“Luca”,”Neri”,1.81,72,”Fisica”);
     p->presentati();
     s->presentati();
}
```

si ottiene che `p->presentati()` esegue il metodo di persona, ma `s->presentati()` esegue quello della classe *studente*, nonostante che la variabile *s* sia di tipo puntatore a *persona*. In pratica quando un metodo è virtuale, allora invocando il metodo tramite un puntatore la versione del metodo da eseguire viene decisa in fase di esecuzione in base al tipo dell'oggetto puntato.

Come ulteriore esempio, forse più convincente

```
int main() {
     string scelta;
     persona *p;
     cout << “persona o studente ? “;
     cin >> scelta;
     if(scelta==”persona”)
          p=new persona(“Carla”,”Rossi”,1.70,56);
     else
          p=new studente(“Luca”,”Neri”,1.81,72,”Fisica”);
     p->presentati();
}
```

A seconda di cosa sceglie l'utente (*studente* o *persona*), il puntatore *p* conterrà l'indirizzo di una persona o di uno studente e il metodo *presentati* si comporterà nel modo più consono.

La variabile *p* è detta **polimorfa**. Il polimorfismo è quella situazione in cui una singola variabile o una collezione (array, vector, liste, ecc.) può contenere oggetti di classi diverse (ma analoghe).

Ad esempio

```
class figura {
public:
     virtual double area() { return 0; }
};

class rettangolo : public figura {
     double base, altezza;
public:
     rettangolo(double b,double a) {
```

```
          base=b;
          altezza=a;
     }
     virtual double area() {
          return base*altezza;
     }
};
class triangolo : public figura {
     double base, altezza;
public:
     triangolo(double b,double a) {
          base=b;
          altezza=a;
     }
     virtual double area() {
          return 0.5*base*altezza;
     }
};
class cerchio : public figura {
     double raggio;
public:
     cerchio(double r) {
          raggio=r;
     }
     virtual double area() {
          return M_PI*raggio*raggio;
     }
};
int main() {
```

```
    figura *f[4];
    f[0]=new cerchio(5);
    f[1]=new rettangolo(4,6);
    f[2]=new rettangolo(5,11);
    f[3]=new triangolo(6,9);
    double totale=0;
    for(int i=0;i<4;i++)
        totale = totale + f[i]->area();
    cout << “area totale “ << totale << endl;
}
```

L'array *f* è un array di puntatori ad oggetti di tipo *figura*, ma poi ogni elemento di *f* contiene l'indirizzo di oggetti delle varie sottoclassi di *figura*. Essendo virtuale il metodo *area*, la chiamata

```
f[i]->area()
```

viene risolta, a tempo di esecuzione, elemento per elemento, invocando quindi, per ognuno di essi, il metodo corretto di calcolo dell'area.

Il polimorfismo è uno strumento importante nella programmazione orientata agli oggetti ed è infatti alla base di molti dei cosiddetti *design pattern.*

Ridefinizione degli operatori

Un aspetto molto interessante del C++ è la possibilità di ridefinire gli operatori (**overloading**) quali +, -, *, ecc. In realtà la ridefinizione è un'estensione, nel senso che la nuova definizione dell'operatore rende possibile usare operandi di un tipo non previsto in precedenza, ma in tutte le altre situazioni l'operatore si comporta come al solito.

Ad esempio è possibile ridefinire l'operatore + per i numeri complessi gestiti con la classe definita nel capitolo precedente, ma in tutti gli altri casi (numeri interi, numeri reali, stringhe) l'operatore + svolge il suo usuale compito.

Quando si ridefinisce un operatore si deve tenere conto che non si possono cambiare

- il numero di argomenti dell'operatore, ad esempio è vietato scrivere un operatore * con tre argomenti
- la posizione degli argomenti rispetto all'operatore, ad esempio è vietato scrivere un operatore –

postfisso, del tipo x-

- la precedenza dell'operatore rispetto agli altri operatori
- l'associatività, ovvero se l'operatore è associativo a sinistra o a destra.

Inoltre non si può definire un operatore non esistente in C++ (ad esempio ** usato in molti linguaggi per indicare l'elevamento a potenza).

E' possibile ridefinire molti altri operatori, ad esempio gli operatori di confronto (==, !=, <, ecc.), aritmetici (+,-,*,/) ma anche gli operatori di assegnamento (=, +=, ecc.).

E' possibile ridefinire operatori speciali quali l'operatore =, usato per l'assegnamento, l'operatore (), che consente di vedere un oggetto come se fosse una funzione, e l'operatore [], che permette di trattare un oggetto alla stregua di un array.

Esistono due modi distinti di considerare un operatore binario (cioè con due argomenti). Il primo è vedere l'operatore come funzione. Quindi

```
a+b
```

corrisponde a

```
operator+(a,b)
```

Il secondo modo è di vederlo come metodo, quindi corrispondente a

```
a.operator+(b)
```

Questa duplice visione è possibile anche per operatori unari:

```
!b
```

può corrispondere a

```
operator!(b)
```

oppure a

```
b.operator!()
```

Le due visioni dell'operatore, seppur analoghe, hanno dei punti di differenza. Innanzitutto un operatore visto come funzione potrebbe aver bisogno di accedere alla parte privata dei suoi operandi: in tal caso va dichiarato “amico” della classe dei suoi operandi.

Invece è possibile ridefinire un operatore visto come metodo solo se si ha disposizione il codice della classe.

Ad esempio definiamo un operatore “+” tra numeri complessi.

```
class complesso {
     double re, im;
```

```
public:
     complesso();
     complesso(double x,double y);
     void scrivi();
     double modulo();
     complesso operator+(const complesso &c);
};
```

L'implementazione dell'operatore “+” è abbastanza simile al metodo *somma*

```
complesso complesso::operator+(const complesso &c) {
     complesso r;
     r.re=re+c.re;
     r.im=im+c.im;
     return r;
}

int main() {
     complesso c1(2,3), c2(1,2);
     complesso c3;
     c3=c1+c2;
     c3.scrivi(); cout << endl;
}
```

Quindi `c1+c2` viene interpretato come se fosse `c1.operator+(c2)`.

Un altro esempio di ridefinizione degli operatori è il seguente. Il metodo “scrivi” è scomodo da usare, al suo posto è possibile ridefinire l'operatore “<<”. Però tale metodo non fa parte della classe “complesso”, ma fa parte della classe “ostream”. Poiché non si può modificare tale classe, è indispensabile definire l'operatore come se fosse una funzione e dichiararlo “amico” della classe, in modo che possa accedere alla parte privata.

```
class complesso {
     double re, im;
```

```
public:
     complesso();
     complesso(double x,double y);
     void scrivi();
     double modulo();
     complesso operator+(const complesso &c);
     friend ostream& operator<<(ostream &s,const complesso &c);
};
```

Si noti che l'operatore << ha due parametri, il primo di tipo *ostream* e il secondo di tipo complesso, entrambi passati per riferimento e restituisce come risultato un riferimento a *ostream*.

Obbligatoriamente il risultato deve essere il primo argomento ricevuto, in modo da poter concatenare correttamente le operazioni di scrittura di più dati (del tipo `cout << a << b << c`). Quindi l'operatore ha il seguente codice

```
ostream& operator<<(ostream &s,const complesso &c) {
     s << c.re << "+i" << c.im;
     return s;
}

int main() {
     complesso c1(2,3), c2(1,2);
     complesso c3;
     c3=c1+c2;
     cout << "somma=" << c3 << endl;
}
```

Tale operatore funziona sia con *cout*, sia con altri flusso, ad esempio quello basati sui file.

L'uso di "const" nella definizione del secondo argomento dell'operatore << consente di visualizzare sullo schermo anche oggetti "temporanei" di tipo complesso, cioé oggetti che non sono memorizzati in variabili, ma sono restituiti come risultato di operazioni. Ad esempio è possibile scrivere direttamente

```
cout << c1+c2;
```

Analogamente, è possibile ridefinire >> per leggere i dati di un oggetto da un flusso:

```
istream& operator>>(istream &s,complesso &c) {
     s >> c.re;
     s >> c.im;
     return s;
}


int main() {
     complesso c1,c2;
     cout << “inserisci due numeri complessi “;
     cin >> c1 >> c2;
     complesso c3;
     c3=c1+c2;
     cout << “somma=” << c3 << endl;
}
```

Anche l'operatore >> deve essere dichiarato “amico” della classe complesso, con un'opportuna dichiarazione “friend” inserita nella classe.

In pratica, è possibile dichiarare che un metodo, una classe, una funzione o un operatore sono *amici* di una classe in modo da consentirgli l'accesso alla parte privata della classe.

Operatore di assegnamento e costruttore di copia

In C++ si può definire come copiare un oggetto in un altro. Ciò può avvenire in due situazioni distinte.

La prima è quando si dichiara una variabile *v* di una classe C e la si inizializza con un oggetto *o*, già esistente e appartenente alla stessa classe C. La seconda è quando si assegna ad una variabile *v* di una classe C un oggetto *o*, sempre della stessa classe C.

La differenza tra i due impieghi è molto sottile: nel primo caso il nuovo oggetto deve essere inizializzato tramite l'oggetto *o*. Invece nel secondo caso la variabile *v* già contiene un oggetto, il quale deve essere “cancellato” per fare posto alla copia del nuovo oggetto *o*.

Usualmente il C++ copia un oggetto in una variabile *v* mediante il semplice procedimento della copia “bit per bit”, in cui il contenuto fisico dell'oggetto, in termini di byte della RAM, è ricopiato nella memoria associata alla variabile *v*.

Per alterare questo meccanismo, che potrebbe essere inadeguato in determinate situazioni, si può

definire un costruttore di copia e ridefinire l'operatore di assegnamento.

Il **costruttore di copia** è un costruttore con un solo parametro, un riferimento costante ad un oggetto della stessa classe.

Ad esempio un costruttore di copia per la classe complesso ha il seguente prototipo

```
complesso(const complesso & c);
```

Una possibile implementazione è

```
complesso::complesso(const complesso &c) {
     re=c.re;
     im=c.im;
}
```

Ovviamente tale implementazione è puramente un esempio, dato che in questa situazione il costruttore di copia fa esattamente una copia esatta di c.

Il costruttore di copia viene invocato automaticamente in casi del genere

```
complesso c3=c2;
```

ma anche quando viene passato un oggetto per valore ad una funzione o ad un metodo.

L'**operatore di assegnamento** è l'operatore = definito come metodo: esso deve copiare un altro oggetto in *this* e restituire come risultato il riferimento all'oggetto stesso.

Ad esempio si dichiara (nella classe) con

```
complesso& operator=(const complesso &c);
```

e si implementa con

```
complesso& complesso::operator=(const complesso &c) {
     re=c.re;
     im=c.im;
     return *this;
}
```

Anche questa implementazione è puramente un esempio, perché effettua una copia fisica dell'oggetto c.

Una situazione in cui il costruttore di copia e l'operatore di assegnamento sono utili è quando un oggetto contiene vector, liste, mappe o comunque oggetti allocati sullo heap.

Esercizi

1. Riscrivere la classe *frazione* (esercizio 2 del cap. precedente), ridefinendo gli operatori matematici +,-,*,/ e gli operatori di lettura >> e scrittura <<
2. Riscrivere la classe *matrice2* (esercizio 5 del cap. precedente), ridefinendo gli operatori matematici +,* e gli operatori di lettura >> e scrittura <<
3. Definire una classe *vettore3* che rappresenta i vettori di $\mathbb{R}^3$ e che ha i seguenti metodi
   a) costruttore per default: crea il vettore nullo
   b) costruttore con tre parametri x,y,z
   c) operatore <<, che scrive le componenti col formato (x,y,z)
   d) operatore >>, che legge le tre componenti
   e) operatore +, somma due vettori
   f) operatore *, calcola il prodotto scalare
   g) operatore ^, calcola il prodotto vettore
   h) double norma(), calcola la norma
4. Definire una classe *polinomio* che rappresenta un polinomio a coefficienti reali e che ha i seguenti metodi
   a) costruttore per default, crea un polinomio nullo
   b) costruttore con un parametro, crea un polinomio costante
   c) costruttore con due parametri, crea un polinomio di grado 1
   d) operatore <<, scrive il polinomio in modo leggibile
   e) operatore >>, legge il grado e poi tutti i coefficienti
   f) void valuta(double x), valuta il polinomio nel punto x
   g) int grado(), restituisce il grado del polinomio
   h) polinomio deriva(), restituisce la derivata prima del polinomio (è un polinomio a sua volta)
5. Definire una classe *orologio_esteso* a partire dalla classe *orologio* (es. 1 del cap. precedente) che gestisce anche la data e ha in più seguenti metodi
   a) costruttore, crea un orologio esteso alla mezzanotte del primo gennaio 2000
   b) void avanza(), avanza di un minuto primo, gestendo in qualche modo il cambio di data e segnalando se la sveglia suona

c) void avanza_giorno(), avanza di un giorno

d) operatore <<, visualizza la data e l'ora correnti

e) void metti_sveglia(int h,int m), mette la sveglia alle ore h e minuti m

f) void disabilita_sveglia()

Suggerimento: orologio_esteso deve accedere ai campi di orologio, quindi o tali campi sono definiti come protected in orologio, oppure sono previsti dei metodi di accesso del tipo dimmi_ora() e dimmi_minuti().

6. Definire una classe *distribuzione* che rappresenta una generica funzione di distribuzione f(x) avente due metodi pubblici virtuali

a) double valuta(double x), che calcola f nel punto x

b) void descrivi(), che descrive il tipo di distribuzione

Definire poi una serie di sottoclassi di *distribuzione*:

1. *normale*, con parametri mu e sigma
2. *esponenziale*, con parametro lambda
3. *uniforme*, con parametri a e b (si intende la distribuzione uniforme nell'intervallo [a,b])

ognuna delle quali deve implementare i metodi virtuali.