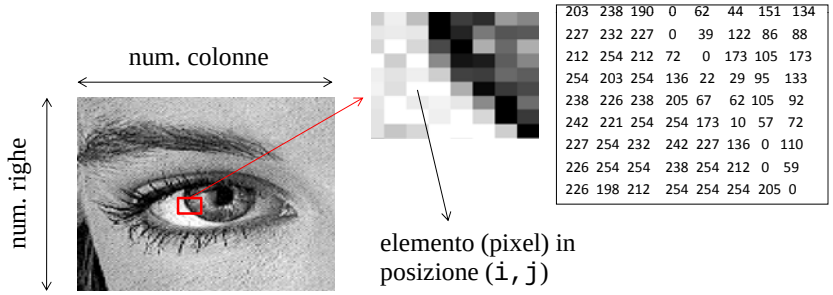
Sommario

- Tipo di dato astratto per descrivere un'immagine

- Programmazione orientata agli oggetti, OOP:
  - Incapsulamento.
  - Composizione.
  - *Ereditarietà*
  - *Polimorfismo.*

Immagini

- Un'immagine è rappresentabile in memoria come una tabella di numeri.
- Ogni pixel può assumere un valore compreso tra 0 e 255. Tale valore corrisponde al valore di intensità luminosa. (0 nero – 255 bianco).



Immagini

- Le immagini sono salvate su disco in diversi formati, che differiscono fra loro a seconda del modo in cui i dati sono memorizzati su file.
- La modalità con cui vengono salvati i dati è definita all'inizio del file (*header*).
- Pertanto un file immagine sarà composto da:
  - *Header* (specifico per ogni formato): contiene informazioni sull'immagine tra cui numero di righe e numero di colonne.
  - *Dati*: valori assunti dai pixel dell'immagine. I valori possono essere salvati in formato ASCII o binario, compresso o non compresso.

Immagini: formato PGM

- Formato utilizzato per memorizzare immagini in scala di grigio.
- L'header è composto nel modo seguente:
  - Intestazione (P2 o P5) a seconda che il file sia in formato ASCII o binario.
  - Numero di colonne e numero di righe.
  - Valore massimo dei livelli di grigio.
  - Possono essere presenti commenti preceduti da #.
- I dati sono memorizzati in formato ASCII (se P2) o binario (se P5) ordinati per righe.

Immagini

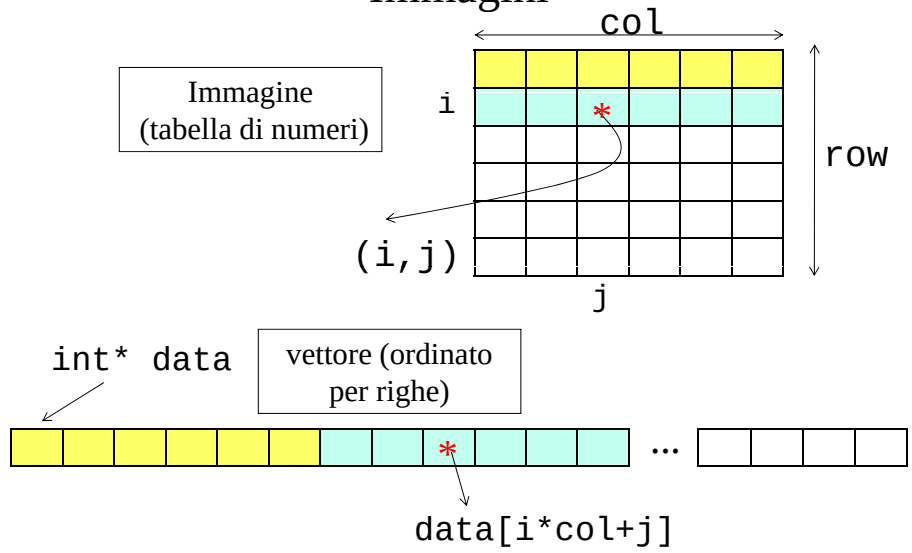
- E' possibile definire un tipo di dato astratto per descrivere un'immagine.
- I pixels dell'immagine sono memorizzati in un *vettore*, ordinati per righe.

MyImage.h

```
class MyImage{

        string type;
        int row;
        int col;
        int maxval;
        int * data;
public:

        CMyImage();
        CMyImage(int r, int c);
        CMyImage(string nomefile);
        ~CMyImage();

        bool Open(string nomefile);
        bool Save(string nomefile);

        ...
};
```

altri metodi necessari alla gestione di un'immagine

Immagini



Operatori sovraccaricati: ()

– Si può eseguire l'overloading dell'operatore di chiamata a funzione (), creando in tal modo una funzione operator alla quale è possibile passare qualsiasi tipo e numero di parametri e che restituisce un valore di qualsiasi tipo: `operator()(...)`

Operatori sovraccaricati: () - Immagine

- Si può eseguire l'overloading dell'operatore di chiamata a funzione (), per la classe MyImage per accedere ai pixels dell'immagine.

Il tipo di ritorno è un riferimento, in modo tale da poter utilizzare l'operatore sia sul *lato sinistro* sia sul *lato destro* di un assegnamento.

```
int& CMyImage::operator ()(int i, int j)
{
        return data[i*col+j];
}
```

Operatori sovraccaricati: () - Immagine

- Pertanto si potranno leggere e modificare i pixels dell'immagine nel modo seguente:

```
MyImage im1("eye.pgm");
```

– Utilizzo dell'operatore () sul *lato sinistro* di un assegnamento:

```
im1(10,15)=0;
```

– Utilizzo dell'operatore () sul *lato destro* di un assegnamento:

```
int val=im1(23,24);
```


Programmazione orientata agli oggetti

- La programmazione orientata agli oggetti (*object-oriented programming*, OOP) si basa su tre principi fondamentali:
  – Incapsulamento (*encapsulation*): come nascondere l'implementazione di una classe.
  – Ereditarietà (*inheritance*): come promuovere un corretto ri-uso del codice.
  – Polimorfismo (*polymorphism*): come manipolare tipi (classi) tra loro collegati in un modo uniforme.

OOP: incapsulamento

- L'incapsulamento permette di nascondere i dettagli di implementazione non necessari all'utente della classe: per esempio, il programmatore utente vuole leggere due immagini, farne la differenza e salvare il risultato:

```
MyImage a("im1.pgm");
MyImage b("im2.pgm");
MyImage res = a-b;
res.Save("diff.pgm");
```


OOP: incapsulamento

- La classe `MyImage` ha incapsulato i dettagli interni di leggere un'immagine dall'hard disk, caricala in memoria, elaborarla e salvare il risultato nuovamente in un file.
- L'OOP permette al programmatore utente della classe di scrivere codice in modo più semplice: non deve preoccuparsi del codice per la gestione delle immagini.
- Si devono solo creare *istanze* (oggetti) e spedire gli appropriati *messaggi* (metodi).

OOP: incapsulamento

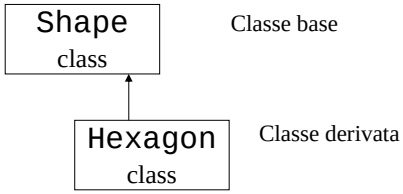
- Un altro aspetto dell'incapsulamento è la protezione dei dati: i campi di una classe (lo stato di un oggetto) devono essere dichiarato `private` e non `public`.
- In tal modo gli utenti della classe devono utilizzare dei metodi per poter leggere o scrivere i dati privati di un oggetto: non si accede mai direttamente all'implementazione interna di una classe.

OOP: ereditarietà

- L'ereditarietà permette di costruire nuove definizioni di classe utilizzando definizioni di classi esistenti.
- L'ereditarietà permette di estendere il comportamento di una *classe base* (base class or parent class), abilitando una *classe derivata* (derived class or child class) ad ereditare i metodi e i campi di tale classe base.

OOP: ereditarietà

- L'ereditarieta classica descrive una relazione del tipo *"è un"* (*"is-a"*).



- Il diagramma si legge come "*un esagono è una forma*": quando le classi sono legate da una relazione del tipo "is-a" allora sono in relazione di ereditarietà.

OOP: ereditarietà

- In questo caso la classe `Shape` descrive campi e metodi comuni a tutte le forme geometriche e quindi è la classe base (quella più generale).
- La classe `Hexagon` è una (is-a) forma, quindi eredita tutti i campi e metodi di `Shape` (*senza doverli riscrivere*) e poi definisce i propri specifici metodi e campi (è una classe derivata, meno generale).

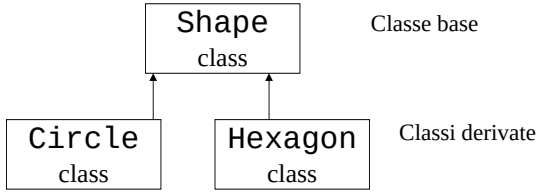
OOP: composizione

- Un'altra forma di riuso del codice nella OOP è la *composizione* (una relazione del tipo "*ha un*", "*has-a*"): alcuni campi della classe sono oggetti di altre classi.
- In questo caso non ci sono relazioni del tipo "is-a", come tra classe base e classi derivate.
- Si ricorda l'esempio della classe `Line` che rappresenta una linea che passa tra due punti del piano: la linea "has-a" punto e non "is-a" un punto, quindi è composizione.

OOP: polimorfismo

- Il polimorfismo permette di trattare nello stesso modo oggetti di classi diverse ma che sono collegate tra loro attraverso l'ereditarietà.
- La classe base definisce un insieme di metodi (interfaccia polimorfica) che sono comuni a tutte le classi derivate.
- L'interfaccia polimorfica è basata su *metodi virtuali* che ogni classe derivata può modificare per renderli specifici.
- Attraverso l'interfaccia polimorfica ogni oggetto risponde in modo specifico alla stessa richiesta.

OOP: polimorfismo



- Per esempio, la classe `Shape` definisce un *metodo virtuale* `Draw()` e tutte le sue classi derivate lo ridefiniscono (*overriding*), perché ogni forma si disegna in un modo diverso.

OOP: polimorfismo

- Allora attraverso un puntatore della classe base posso invocare il metodo corretto per ogni oggetto creato:

```
void elabora(Shape *p){
   //...
   p->Draw();
}
int main(){
   Circle c;
   Hexagon h;

   elabora(&c);
   elabora(&h);
   return 0;
}
```

Disegna un cerchio o un esagono, anche se è invocato da un puntatore a `Shape`