Object Oriented Programming

- Basata sul costrutto fondamentale **classe**.
  - Tipo di dato che contiene anche codice
  - Ovvero funzioni specifiche per quel tipo di dato
  - Permette di incapsulare dati e codice per gestire **oggetti** del programma.
- Gli oggetti sono istanze delle classi
- **Idea di base**: combinare le descrizioni degli elementi (i dati) con le azioni eseguibili su quegli elementi (le funzioni)
- Classi e Oggetti sono gli elementi chiave su cui poggia l'OOP

object oriented programming

- Nel mondo reale un oggetto è qualcosa che possiamo percepire.
  - Ha proprietà specifiche (colore, forma ecc)
  - Può avere anche un comportamento

- In OOP un **oggetto** è qualsiasi cosa nella quale si possano immagazzinare
  - dati
  - operazioni per manipolare tali dati

object oriented programming

- Gli oggetti comunicano fra loro mediante "messaggi" che includono
  - l'identificatore dell'azione che l'oggetto destinatario deve eseguire,
  - i dati che saranno necessari all'oggetto per eseguire l'azione

- L'utente comunica con l'oggetto tramite la sua interfaccia.
  - Insieme di operazioni definite dalla classe dell'oggetto in modo che siano visibili al programma

- Interfaccia: vista semplificata dell'oggetto
  - non si sa com'è fatto ma lo si sa usare mediante la sua interfaccia

Esempio

- Programma gestionale di un negozio
  - Un oggetto per ogni articolo gestito dal negozio
  - Il programma può gestire e manipolare gli stessi dati per più oggetti
    - Es: costo, numero unità, data nuovo ordine
  - Gli oggetti possono effettuare azioni con i propri dati
    - Es: inizializzare i propri dati, modificare i propri campi

- Un oggetto consiste dunque di dati e azioni.

le classi

- Classe: insieme di oggetti che condividono una struttura ed un comportamento
- Contiene la specifica dei dati che descrivono l'oggetto che ne fa parte, insieme alla descrizione delle azioni che l'oggetto stesso è capace di eseguire
- in C++ questi dati si denominano *attributi* o *variabili*, mentre le azioni si dicono *funzioni membro* o *metodi*

- Le classi definiscono tipi di dato personalizzati in funzione dei problemi da risolvere,
  - ciò facilita scrittura e comprensione delle applicazioni;
- Possono separare l'interfaccia dall'implementazione;
  - solo il programmatore della classe conoscerà i dettagli implementativi,
  - l'utilizzatore deve soltanto conoscere l'interfaccia

definizione di una classe

- Due parti:
  - *dichiarazione*: descrive i dati e l'interfaccia (cioè le "funzioni membro", anche dette "metodi")
  - *definizioni dei metodi*: descrive l'implementazione delle funzioni membro

```
class NomeClasse          // Identificatore valido
{
  dichiarazioni dei dati        // attributi
  definizione delle funzioni    // metodi
};
```

- Attributi: variabili semplici (interi, strutture, arrays, float, ecc.)
- Metodi: funzioni semplici che operano sugli attributi (*dati*)

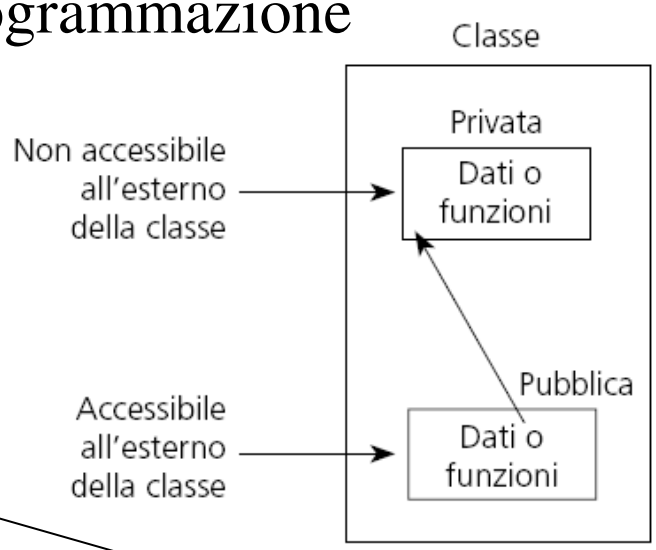
specificatori di accesso

- Per default, i membri di una classe sono nascosti all'esterno, cioè, i suoi dati ed i suoi metodi sono *privati*
- E' possibile controllare la *visibilità* esterna mediante specificatori d'accesso:
  - la sezione `public` contiene membri a cui si può accedere dall'esterno della classe
  - la sezione `private` contiene membri ai quali si può accedere solo dall'interno
  - ai membri che seguono lo specificatore `protected` si può accedere anche da metodi di classi *derivate* della stessa

```
class NomeClasse
{
   public:
      Sezione pubblica    // dichiarazione di membri pubblici
   protected:
      Sezione protetta    // dichiarazione di membri protetti
   private:
      Sezione privata     // dichiarazione di membri privati
};
```


information hiding / incapsulamento

- Questa caratteristica della classe si chiama *occultamento di dati (information hiding)* ed è una proprietà dell'OOP
  - tecnica nota anche come *incapsulamento*,
  - limita molto gli errori rispetto alla programmazione strutturata




```
class Semaforo
{
   public:
      void cambiareColore();
      //...
   private:
      enum Colore {VERDE, ROSSO, GIALLO};
      Colore c;
};
```

Interfaccia

Classi vs strutture

- Potremmo definire classi anche usando `struct`
- Tutti i membri sarebbero visibili dall'esterno
  - Per mantenere compatibilità con C

regole pratiche

- le dichiarazioni dei metodi (i.e. intestazioni delle funzioni), normalmente, si collocano nella sezione pubblica
- le dichiarazioni dei dati (attributi), normalmente, si mettono nella sezione privata

- E' indifferente collocare prima la sezione pubblica o quella privata;
  - Meglio collocare la sezione pubblica prima per mettere in evidenza le operazioni che fanno parte dell'interfaccia utente pubblica

- `public` e `private` seguite da due punti, segnalano l'inizio delle rispettive sezioni pubbliche e private;
  - una classe può avere varie sezioni pubbliche e private

- L'interfaccia deve essere pubblica
  - Altrimenti non può essere invocata dal programma!


```
class Prova
{
   private:
      float costo;
      char nome[20];
   public:
      void calcolare(int);
};
```

Private non è necessario
- E' utile per evidenziare l'occultamento

Per usare una classe bisogna sapere
- Nome
  - Tipicamente definito in un header file (con lo stesso nome della classe)
- Dove è definita
- Che operazioni supporta

- Nelle definizioni delle classi si collocano (tipicamente) solo le intestazioni dei metodi
  - Le definizioni stanno in un file di *implementazione* (estensione .cpp)

oggetti

- definita una classe, possono essere generate *istanze* della classe, cioè *oggetti*

nome_classe identificatore ;

```
rettangolo r;
Semaforo S;
```

- un oggetto sta alla sua classe come una variabile al suo tipo
- quello che nelle `struct` era l'operatore di accesso al campo (`.`), qui diventa l'operatore *di accesso* al membro

```
Punto p;
p.Fissarex (100);
cout << " coordinata x è " << p.Leggerex();
```


oggetti

- Gli oggetti (come le strutture) possono essere copiati
- C++ fa una copia bit a bit di tutti i membri
  - Tutti i membri presenti nell'area dati dell'oggetto originale vengono copiati nell'oggetto destinatario

```
rettangolo r1;
...
rettangolo r2;

r2=r1;
```


dati membro

- possono essere di qualunque tipo valido,
  - tranne il tipo della classe che si sta definendo
- non è permesso inizializzare un membro dato di una classe all'atto della sua definizione;
  - la seguente definizione genera errori:

```
class C {
private:
   int T = 0;                    // Errore
   const int CInt = 25;          // Errore
   int& Dint = T                 // Errore
// ...
};
```

- la definizione della classe indica semplicemente il *tipo* di ogni membro dato (non riserva realmente memoria)
- si deve invece inizializzare i membri dato ogni volta che si crea un'*istanza specifica* della classe mediante il *costruttore* della classe

funzioni membro

- possono essere sia dichiarate che definite all'interno delle classi; la definizione consiste di quattro parti:
  - il tipo restituito dalla funzione
  - il nome della funzione
  - la lista dei parametri formali (eventualmente vuota) separati da virgole
  - il corpo della funzione racchiuso tra parentesi graffe

- le tre prime parti formano il **prototipo** della funzione che *deve essere definito* dentro la classe,
  - il corpo della funzione può essere definito altrove

```
class Articolo_Vendite {
public:
 double prezzo_medio(); // dichiarazione prototipo (definito altrove)
 bool articolo_uguale (const Articolo_Vendite & art) // definizione
   {return iva == art.iva; }                         // funzione
private:   // membri privati
   // ...
};
```


funzioni membro

- La definizione di funzioni dichiarate in una classe deve contenere il riferimento alla classe

```
tipo_restituito Nome_Classe :: Nome funzione (lista
parametri)
{
   corpo della funzione
}
```


funzioni membro costante

- La funzione non può cambiare gli attributi dei suoi oggetti

Sintassi:

```
tipo_restituito Nome_Classe :: Nome funzione (lista
parametri) const
{
   corpo della funzione
}
```


chiamate a funzioni membro

- i metodi di una classe s'invocano così come si accede ai dati di un oggetto, tramite l'operatore punto (.) con la seguente sintassi:

nomeOggetto.nomeFunzione (valori dei parametri)

```
class Demo
{
private:
   // ...
public:
   void funz1 (int P1)
      {...}
   void funz2 (int P2)
      {...}
};
Demo d1, d2;       // definizione degli oggetti d1 e d2
...
d1.funz1(2005);
d2.funz1(2010);
```

Alcuni linguaggi chiamano messaggi le invocazioni a funzioni membro

Tipi di funzioni membro

- Costruttori e distruttori
  - Invocati automaticamente alla creazione e alla distruzione di oggetti
- Selettori
  - Restituiscono valori di membri dato
- Modificatori
  - Modificano i valori di membri dato
- Operatori
  - definiscono operatori standard in C++
- Iteratori
  - elaborano collezioni di oggetti (es. array)

funzioni *inline* e *offline*

- i metodi definiti nella classe sono funzioni in linea; per funzioni grandi è preferibile codificare nella classe solo il prototipo della funzione
- nella definizione *fuori linea* della funzione bisogna premettere il nome della classe e l'*operatore di risoluzione di visibilità* `::`;

```
class Punto {
public:
   void FissareX(int valx);
private:
   int x;
   int y;
};
...
void Punto::FissareX(int valx)
{
   // ...
}
```


header files ed intestazioni di classi

- il codice sorgente di una classe si colloca normalmente in un file indipendente con lo stesso nome della classe ed estensione
  - `.cpp`

- le dichiarazioni si collocano normalmente in header files indipendenti da quelli che contengono le implementazioni dei metodi

costruttori

- Può essere conveniente che un oggetto si possa auto-inizializzare all'atto della sua creazione, senza dover effettuare una successiva chiamata ad una sua qualche funzione membro
- un *costruttore* è un metodo di una classe che viene automaticamente eseguito all'atto della creazione di un oggetto di quella classe
- ha lo stesso nome della propria classe e può avere qualunque numero di parametri ma non restituisce alcun valore (neanche `void`)

```
class Rettangolo
{
   private:
      int Sinistro;
      int Superiore;
      int Destro;
      int Inferiore;
   public:
      Rettangolo(int Sin, int Sup, int Des, int Inf); // Costruttore
      // definizioni di altre funzioni membro
};
```


definizione oggetto con costruttore

- quando si definisce un oggetto, si passano i valori dei parametri al costruttore utilizzando la sintassi di una normale chiamata di funzione:

```
Rettangolo rect(25, 75,25,75); // rect è ISTANZA di Rettangolo

Rettangolo* nr = new Rettangolo(25, 75,25,75); // nr punta
                                  // una nuova ISTANZA di Rettangolo
```

- un costruttore senza parametri si chiama *costruttore di default*;
  - inizializza i membri dato assegnandogli valori di default
- C++ crea automaticamente un costruttore di default quando non vi sono altri costruttori,
  - esso non inizializza i membri dato della classe a valori predefiniti
- un *costruttore di copia* è creato automaticamente dal compilatore quando
  - si passa un oggetto per valore ad una funzione (si costruisce una copia locale dell'oggetto)
  - si definisce un oggetto inizializzandolo ad un oggetto dello stesso tipo

distruttore

- si può definire anche una funzione membro speciale nota come *distruttore*, che viene chiamata automaticamente quando si distrugge un oggetto
- il distruttore ha lo stesso nome della sua classe preceduto dal carattere ~
- neanche il distruttore ha tipo di ritorno ma, al contrario del costruttore, non accetta parametri e *non* ve ne può essere più d'uno

```
class Demo
{
private:
   int dati;
public:
   Demo()  {dati = 0;}          // costruttore
   ~Demo() {}                   // distruttore
};
```

- serve normalmente per liberare la memoria assegnata dal costruttore
- se non si dichiara esplicitamente un distruttore, C++ ne crea automaticamente uno vuoto

sovraccaricamento di metodi

- anche le funzioni membro possono essere sovraccaricate,
  - ma soltanto nella loro propria classe,
- Seguono le stesse regole utilizzate per sovraccaricare funzioni ordinarie
  - due funzioni membro sovraccaricate non possono avere lo stesso numero e tipo di parametri
- l'*overloading* permette di utilizzare uno stesso nome per più metodi che si distingueranno solo per i parametri passati all'atto della chiamata

```
class Prodotto
{
public:
   int prodotto (int m, int n);                  // metodo 1
   int prodotto (int m, int p, int q);           // metodo 2
   int prodotto (float m, float n);              // metodo 3
   int prodotto (float m, float n, float p);     // metodo 4
}
```

