Sommario

- Classi e oggetti, un esempio:
  - Oggetti in memoria *stack* e in memoria *heap*
  - Oggetti e funzioni
  - Oggetti ed array
- Il puntatore implicito `this`
- Costruttore: Lista di inizializzazione dei membri
- Tipo di dato astratto: *Abstract DataType* (ADT).

Esempio

- Sviluppiamo un esempio di classe che rappresenta una canzone: per semplicità consideriamo di caratterizzare l'oggetto solo con il titolo e la lunghezza del brano.
- Lo stato dell'oggetto è *privato*, quindi si devono inserire metodi per modificare/scrivere lo stato (*Set*) e metodi per ottenere/leggere lo stato (*Get*). Inoltre si inserisce un metodo per stampare a monitor lo stato dell'oggetto.
- Si inserisce il costruttore di default per mezzo degli argomenti di default.

Esempio: Canzone.h

```
#ifndef CANZONE_H
#define CANZONE_H

#include <string>
using namespace std;

class Canzone{
   string titolo;
   double durata;
public:
   Canzone(string tit="", double dur=0);
   ~Canzone();
   void SetTitolo(string tit);
   void SetDurata(double dur);
   string GetTitolo();
   double GetDurata();
   void Stampa();
};
#endif
```

I campi devono essere privati

Argomenti di default

Esempio: Canzone.cpp

```
#include "Canzone.h"
#include <iostream>
#include <string>
using namespace std;
Canzone::Canzone(string tit, double dur){
   titolo=tit;
   durata=dur;
   cout<<"Costruttore("<<titolo<<",
   "<<durata<<")"<<endl;
}
Canzone::~Canzone(){
   cout<<"Distruttore("<<titolo<<",
   "<<durata<<")"<<endl;
}
void Canzone::SetTitolo(string tit){
   titolo=tit;
}
void Canzone::SetDurata(double dur){
   if (dur>=0)
         durata=dur;
   else
         cout<<"Durata negativa!"<<endl;
}
```

I metodi possono accedere direttamente ai campi privati

Modificare lo stato attraverso dei metodi permette di eseguire controlli

```
string Canzone::GetTitolo()
{
   return titolo;
}
double Canzone::GetDurata()
{
   return durata;
}
void Canzone::Stampa()
{
   cout<<titolo<<":
   "<<durata<<endl;
}
```

Esempio: Test.cpp

```
#include "Canzone.h"
#include <iostream>
using namespace std;

void test1();
void test2();
void test3();

void modifica(Canzone *c);

int main(){
   test1();
   test2();
   test3();
   return 0;
}
```

Puntatore al tipo (classe) `Canzone`, quindi un passaggio per riferimento

Esempio: test1()

- In questo test viene creato un oggetto in memoria *stack* (`c1`) e un oggetto in memoria *heap* (`c2`), cioè un'allocazione dinamica, che quindi deve essere esplicitamente deallocato con `delete`.
- Sono evidenziate le chiamate ai costruttori e distruttori.
- Viene verificato il controllo sull'inserimento di una durata negativa.
- Viene mostrato come invocare un metodo su un oggetto e su un puntatore ad un oggetto.

Esempio: test1()

```
void test1(){
  Canzone c1("She Wolf",3.07);
  Canzone *c2 = new Canzone("Diamonds",5.40);

  cout<<"--------------"<<endl;

  c1.SetDurata(-3.1);
  c1.Stampa();
  c2->Stampa();

  cout<<"--------------"<<endl;

  delete c2;
}
```

Allocazione dinamica di un oggetto

Una possibile uscita

```
Costruttore(She Wolf, 3.07)
Costruttore(Diamonds, 5.4)
----------------------
Durata negativa!
She Wolf: 3.07
Diamonds: 5.4
----------------------
Distruttore(Diamonds, 5.4)
Distruttore(She Wolf, 3.07)
```


Esempio: test2()

- In questo test sono creati degli oggetti utilizzando il costruttore di default, inoltre sono evidenziate le chiamate ai costruttori e distruttori.
- Viene evidenziato il *passaggio per riferimento* di 'un oggetto' e 'un puntatore ad un oggetto' ad una funzione (quindi senza creare una copia locale di tale oggetto nella funzione), in modo tale che le modifiche apportate all'oggetto nella funzione chiamata siano valide anche nella funzione chiamante.

Esempio: test2()

```
void test2(){
   Canzone c1;
   Canzone *c2 = new Canzone();
   modifica(&c1);
   modifica(c2);
   c1.Stampa();
   c2->Stampa();
   delete c2;
}
```

Una possibile uscita

```
Costruttore(, 0)
Costruttore(, 0)
Inserire titolo e durata: prima 2.5
Inserire titolo e durata: seconda 3.4
prima: 2.5
seconda: 3.4
Distruttore(seconda, 3.4)
Distruttore(prima, 2.5)
```

```
void modifica(Canzone *c){
   cout<<"Inserire titolo e durata: ";
   string tit;
   double dur;
   cin>>tit>>dur;
   c->SetTitolo(tit);
   c->SetDurata(dur);
}
```


Esempio: test3()

- In questo test viene creato dinamicamente un array di oggetti e quindi utilizzato il costruttore di default. Poi sono inserite le caratteristiche di ogni oggetto utilizzando i metodi per scrivere/modificare (*Set*) lo stato.
- Sono usate le funzioni per leggere/ottenere (*Get*) lo stato degli oggetti al fine di poter stampare solo una porzione dello stato e per poter utilizzare i valori dello stato per fare delle operazioni.

Esempio: test3()

```
void test3(){
   int dim=3;
   Canzone *playlist=new Canzone[dim];
   playlist[0].SetTitolo("Diamonds");
   playlist[0].SetDurata(4.49);
   playlist[1].SetTitolo("She Wolf");
   playlist[1].SetDurata(3.07);
   playlist[2].SetTitolo("Skyfall");
   playlist[2].SetDurata(4.46);
   cout<<"---------"<<endl;

   cout<<"La playlist e` composta da:"<<endl;
   for(int i=0;i<dim;i++)
        cout<<playlist[i].GetTitolo()<<endl;

   cout<<"La playlist ha una durata di ";
   double sum=0;
   for(int i=0;i<dim;i++)
        sum+=playlist[i].GetDurata();
   cout<<sum<<endl;

   cout<<"---------"<<endl;
   delete [] playlist;
}
```

Una possibile uscita

```
Costruttore(, 0)
Costruttore(, 0)
Costruttore(, 0)
---------
La playlist e` composta da:
Diamonds
She Wolf
Skyfall
La playlist ha una durata di 12.02
---------
Distruttore(Skyfall , 4.46)
Distruttore(She Wolf, 3.07)
Distruttore(Diamonds, 4.49)
```


Il puntatore implicito `this`

- Ogni oggetto mantiene la propria copia dei dati membro della classe.
- Esiste una sola copia di ogni funzione membro di una classe.

- L'associazione del metodo all'oggetto corrente avviene attraverso il puntatore implicito `this`.
- Ogni funzione membro di una classe contiene *un puntatore all'oggetto* tramite il quale il metodo è stato invocato, il puntatore `this`.

Il puntatore implicito `this`

- Pertanto ogni metodo di una classe può accedere direttamente ai campi perché vi è tale puntatore implicito.

- Nella definizione di una funzione membro si può fare riferimento esplicitamente al puntatore `this`, anche se in generale *non* è necessario. Tuttavia in alcuni casi è necessario ricorrere a tale puntatore.

Il puntatore implicito `this`

- Per esempio, il metodo `SetTitolo` può essere scritto in uno dei due modi seguenti:

```
void Canzone::SetTitolo(string tit){
      titolo=tit;
}
```

```
void Canzone::SetTitolo(string tit){
      this->titolo=tit;
}
```

Viene reso esplicito che i membri dell'oggetto corrente (cioè quello su cui è invocato il metodo) sono visibili attraverso il puntatore implicito `this`

Costruttori: lista di inizializzazione dei membri

- Come visto negli esempi, nel corpo del costruttore sono inizializzati i dati membro.

```
Canzone::Canzone(string tit, double dur){
  titolo=tit;
  durata=dur;
  cout<<"Costruttore("<<titolo<<","<<durata<<")"<<endl;
}
```

- Una sintassi alternativa è *la lista di inizializzazione dei membri*. Una lista di nomi di attributi e dei loro valori separati da virgole.

Costruttori: lista di inizializzazione dei membri

- La lista di inizializzazione dei membri può essere specificata solo nella definizione del costruttore.
- È posta tra la lista dei parametri e il corpo del costruttore ed è preceduta da due punti, `:` .

```
Canzone::Canzone(string tit, double dur):
                              titolo(tit), durata(dur)
{
      cout<<"Costruttore("<<titolo<<", "<<durata<<")"<<endl;
}
```

Tipi di dati astratti

- Il processo di scrittura del codice dovrebbe essere preceduto da uno *schema* del programma (applicazione) con le sue specifiche.
- Fin dall'inizio è importante specificare ciascun compito in termini di *ingresso* e *uscita*.
- Il *comportamento* del programma è più importante dei meccanismi che lo realizzano.
- Se è necessario un certo oggetto per realizzare alcuni obiettivi, tale oggetto è specificato in termini delle operazioni (metodi pubblici) che vengono svolte su esso, piuttosto che della sua struttura interna (dati, in generale l'implementazione privata).

Tipi di dati astratti

- Un tipo di dato specificato mediante le operazioni possibili su di esso è detto *tipo di dato astratto* (*Abstract DataType, ADT*).
- In C++ un tipo di dato astratto può far parte di un programma sotto forma di *classe*.
- Gli oggetti sono utilizzati attraverso la loro interfaccia pubblica, cioè i metodi pubblici. Sono descritti solo attraverso il *comportamento*.
- I dati e i metodi vengono definiti dalla classe che realizza (*implementa*) l'interfaccia.

Tipi di dati astratti

- Un ADT è un tipo di dato accessibile attraverso un'interfaccia. Si chiama *client* un programma che usa un ADT (classe) e si chiama *implementazione* una classe che definisce il tipo di dato.
- Il vantaggio risiede nella possibilità di poter sviluppare programmi che si *basano sui comportamenti* degli oggetti e non sulla loro implementazione.

Tipi di dati astratti : esempio

- Un punto è caratterizzato, per esempio, dalle sue coordinate *cartesiane* e *polari*.
- Lo si può definire come ADT specificando la sua interfaccia, cioè l'insieme dei suoi metodi pubblici: un costruttore a partire dalle coordinate cartesiane e i metodi per leggere e scrivere lo stato nelle due forme, cartesiana e polare.

Tipi di dati astratti : esempio

Point.h

```
#ifndef POINT_H
#define POINT_H
class Point{
   double r, t;
public:
   Point(double a=0, double b=0);
   ~Point();
   void SetX(double a);
   void SetY(double a);
   void SetR(double a);
   void SetT(double a);
   double GetX();
   double GetY();
   double GetR();
   double GetT();
};
#endif
```


Tipi di dati astratti : esempio

Point.cpp

```
#include "Point.h"
#include <iostream>
#include <cmath>
using namespace std;
Point::Point(double a, double b){
   r = sqrt(a * a + b * b);
   t = atan2(b, a);
}
Point::~Point(){cout<<"Distruttore"<<endl;}
void Point::SetX(double a){
   double y=r*sin(t);
   r=sqrt(a*a+y*y);
   t = atan2(y, a);
}
void Point::SetY(double a){
   double x=r*cos(t);
   r=sqrt(a*a+x*x);
   t = atan2(a, x);
}
void Point::SetR(double a){ r=a;}
void Point::SetT(double a){ t=a;}
double Point::GetX(){ return r*cos(t);}
double Point::GetY(){ return r*sin(t);}
double Point::GetR(){ return r;}
double Point::GetT(){ return t;}
```

Rappresentazione interna dei dati in coordinate polari.

Tipi di dati astratti : esempio

- La prima funzione di test verifica la correttezza dell'implementazione.
- La seconda funzione di test esegue un *benchmark* per valutare le prestazioni dell'implementazione della classe `Point`. Vengono eseguite molte volte delle operazioni comuni e viene *misurato il tempo di esecuzione*.

Tipi di dati astratti : esempio

```
void test1(){
  Point p1(1,1);
  cout<<"cart: "<<p1.GetX()<<" "<<p1.GetY()<<endl;
  cout<<"pol: "<<p1.GetR()<<" "<<p1.GetT()<<endl;
  p1.SetX(2);
  cout<<"cart: "<<p1.GetX()<<" "<<p1.GetY()<<endl;
  cout<<"pol: "<<p1.GetR()<<" "<<p1.GetT()<<endl;
}
```

Una possibile uscita

```
cart: 1   1
pol: 1.41421    0.785398
cart: 2   1
pol: 2.23607     0.463648
Distruttore
```

Tipi di dati astratti : esempio

```
void test2(){
   clock_t start,finish;

   start=clock();

   Point p1(1.2,3.4);
   for(int i=0;i<1e7;i++){
        p1.SetX(4.1);
        double x=p1.GetX();
   }

   finish=clock();
   cout<<"Tempo di esecuzione  "<<(finish - start)<<" ms"<<endl;
}
```

Per utilizzare i tipi e le funzioni legate al tempo è necessario inserire `#include <ctime>`

Una possibile uscita

```
Tempo di esecuzione    4265 ms
Distruttore
```


Tipi di dati astratti : esempio

- L'interfaccia di un ADT definisce un "*contratto*" tra utenti e implementatori che impiega precisi strumenti di comunicazione fra i due contraenti.
- Sfruttando il concetto di ADT, qualsiasi rappresentazione interna dei dati (cartesiana o polare) non modifica l'uso che ne fanno i client, perchè il comportamento non cambia.

- La ragione di modificare la rappresentazione dei dati è, per esempio, quella di ottenere *prestazioni migliori*.
- In `test2()` (client) dell'esempio sono usate con frequenza `SetX` e `GetX`, pertanto si ottengono prestazioni migliori usando l'implementazione in coordinate cartesiane.

Point.cpp

Tipi di dati astratti : esempio

```
#include "Point.h"
#include <iostream>
#include <cmath>
using namespace std;
Point::Point(double a, double b){
   r = a;
   t = b;
}
Point::~Point(){cout<<"Distruttore"<<endl;}
void Point::SetX(double a){ r=a;}
void Point::SetY(double a){ t=a;}
void Point::SetR(double a){
   double th=atan2(t,r);
   r=a*cos(th);
   t=a*sin(th);
}
void Point::SetT(double a){
   double rh=sqrt(r*r+t*t);
   r=rh*cos(a);
   t=rh*sin(a);
}
double Point::GetX(){ return r;}
double Point::GetY(){ return t;}
double Point::GetR(){ return sqrt(r*r+t*t);}
double Point::GetT(){ return atan2(t,r);}
```

Rappresentazione interna dei dati in coordinate cartesiane, senza cambiare *Point.h*.

Tipi di dati astratti : esempio

- Anche cambiando l'implementazione della classe `Point` il programma client non cambia, perché non è stato modificato il comportamento, cioè l'interfaccia pubblica: quindi si possono eseguire le stesse due funzioni di test.

- Si ottengono gli stessi risultati, ma sono migliorate le prestazioni: con l'implementazione interna dei dati nel formato cartesiano i tempi di esecuzione scendono a 1141 ms.