SOMMARIO

- Programmazione orientata agli oggetti (OOP)
  - Classi
  - Costruttori
  - Metodi
  - Creazione di oggetti

Programmazione object-oriented (OOP)

- Un linguaggio "di basso livello" fornisco solo l'accesso alle istruzione vere e proprie della macchina.
- Un linguaggio "di livello intermedio" da' invece al programmatore un insieme ristretto di strumenti e gli consente di sviluppare per conto proprio dei costrutti di livello piu' alto.
- Il C viene frequentemente definito un linguaggio di programmazione "di livello intermedio".
  - Non tenta di nascondere l'hardware della macchina al programma.
  - Consente di manipolare direttamente bit, byte, indirizzi e porte.
  - Attraverso le *funzioni* si ottengono le funzionalita' di livello piu' alto.

Programmazione object-oriented (OOP)

- Lo sviluppo di applicazioni complesse porta a voler costruire *moduli software* sempre più versatili, che possano essere *riutilizzati* in numerosi progetti.

- I *linguaggi ad oggetti* (OOP, Object Oriented Programming) introducono delle soluzioni sintattiche adatte a scrivere programmi secondo questa filosofia.

Programmazione object-oriented (OOP)

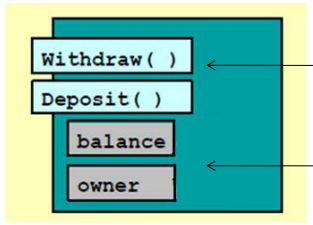
- Tutti i programmi sono composti da due elementi fondamentali: il *codice* e i *dati*.
- Il *codice* e' la parte di programma che svolge le operazioni (ad esempio un algoritmo di ordinamento)
- I *dati* sono i "valori" su cui vengono eseguite le operazioni (ad esempio il vettore di int o double)
- La programmazione orientata agli oggetti permette ai programmatori di definire entita' (gli *oggetti*) caratterizzati da:
  - uno *stato* (un'insieme di dati che descrivono l'oggetto)
  - le *operazioni* (o *metodi*) che possono essere eseguite sull'oggetto o che l'oggetto puo' eseguire.

Programmazione object-oriented (OOP)

- Le strutture (struct) sono estese alle *classi*: si raggruppano i dati e le funzioni che operano su tali dati (*tipo di dato astratto*).
- Le *classi* contengono i *metodi* (l'equivalente delle funzioni), che elaborano i dati contenuti nei *campi*, che costituiscono lo *stato* dell'oggetto.
- Gli *oggetti* sono creati a partire dalle classi.
- Gli *oggetti*, creati (*istanziati)* dalla *classe*, hanno un *tipo*, che è la *classe* dell'*oggetto.*

Programmazione object-oriented (OOP)

Classe :
ContoBancario



Oggetto 1 Oggetto 2

Programmazione object-oriented (OOP)

- Una classe descrive un insieme di oggetti che hanno caratteristiche (dati) e funzionalita' (funzioni) uguali.
- *In tal modo viene definito un nuovo tipo: il programmatore estende il linguaggio definendo i tipi che gli sono necessari alla risoluzione di un determinato problema.*
- Nell'esempio e' stato definito un nuovo tipo: ContoBancario il cui stato e' descritto da:
  - Proprietario (`owner`)
  - Saldo (`balance`)

  E su cui si puo' operare mediante:
  - Depositi (`Deposit(...)`)
  - Prelievi (`Withdraw(...)`)

Programmazione object-oriented (OOP)

I nuovi tipi cosi' definiti si comportano quasi esattamente come i tipi built-in.

- si possono creare variabili di quel tipo (chiamate oggetti o istanze)
- si possono manipolare tali variabili.
- Tutti gli oggetti di una classe avranno le medesime caratteristiche (ad esempio ogni oggetto di tipo ContoBancario avra' un balance).
- Ogni oggetto manterra' la propria individualita' (ogni oggetto contoBancario avra' un valore di balance diverso).

OOP - Accessibilita' dei dati

- Il problema dell'accessibilita' dei dati e' risolto mediante l'*incapsulamento* (*information hiding*).
- In una classe i membri e le funzioni sono racchiusi in una singola unita', che costituisce un insieme chiuso, attraverso il quale si puo' chiaramente definire un interno e un esterno.
- In questo modo e' possibile selezionare cio' che puo' essere accessibile dall'esterno da cio' che non puo' esserlo.
- I dati e i membri accessibili sia dall'interno che dall'esterno sono *public,* quelli accessibili solo dall'interno sono *private.*

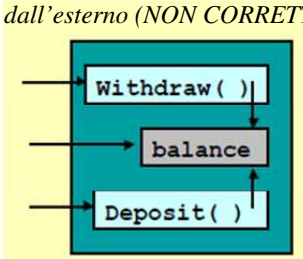
OOP - Accessibilita' dei dati

- I membri pubblici forniscono l'*interfaccia* pubblica della classe (insieme delle operazioni che realizzano il *comportamento* della classe).
- I membri privati rappresentano l'*implementazione* privata, cioè l'insieme dei dati in cui vengono registrate le *informazioni* ed eventuali metodi per la loro manipolazione privata.
- Questa distinzione rispecchia la divisione tra il programmatore della classe e l'eventuale utilizzatore (un altro programmatore).
- I programmi che utilizzano la classe non possono accedere all'implementazione privata, evitando in tal modo di produrre comportamenti indesiderati.

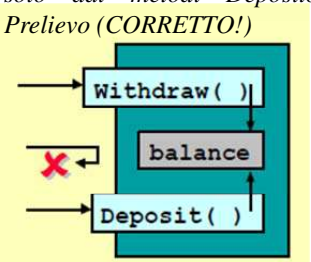
OOP - Incapsulamento

- Ritornando all'esempio del ContoBancario si puo' pensare ai metodi `Deposit` e `Withdraw` come metodi pubblici, al dato membro `balance` come privato.
- In questo modo si puo' modificare `balance` solo attraverso i metodi `Deposit` e `Withdraw` , che, essendo metodi della classe, possono accedere ai campi privati della classe.

Balance puo' essere modificato dall'esterno (NON CORRETTO!)

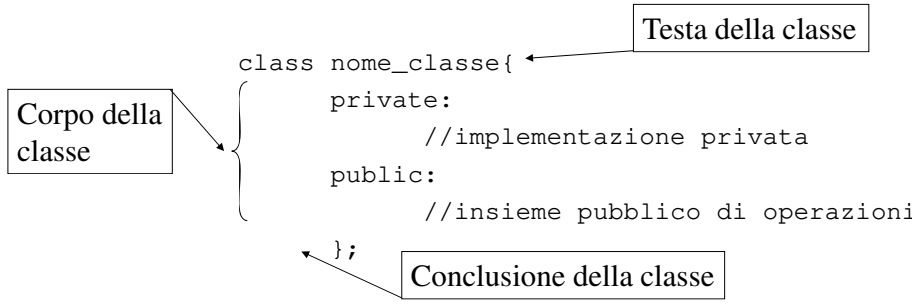


Balance puo' essere modificato solo dai metodi Deposito e Prelievo (CORRETTO!)



CLASSI

- Vediamo la forma generale di una *classe*:

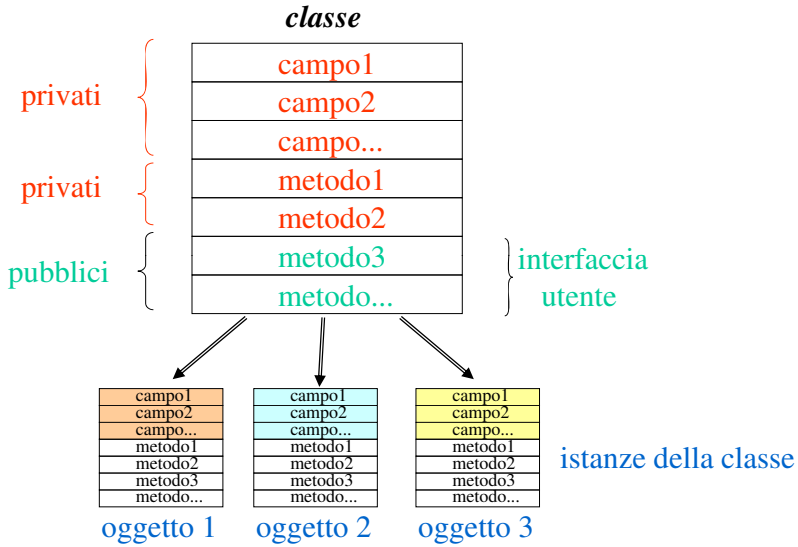


- Le operazioni sono chiamate *funzioni membro* o *metodi*.
- I dati sono chiamati *dati membro* o *attributi*.

CLASSI - Progetto

- Quando si sviluppano programmi utilizzando le classi, e' buona norma dividere il programma in piu' files nel modo seguente:
  - La dichiarazione della classe in un file header che, solitamente, avra' come nome il nome della classe (.h)
  - La definizione dei metodi della classe in un file sorgente, che, solitamente, avra' come nome il nome della classe (.cpp)
- Altre funzioni utili al programma saranno definite in altri files .cpp, cosi' come il main e le funzioni di test.

CLASSI



CLASSI

Sviluppiamo la classe `ContoBancario`.

ContoBancario.h

```
#ifndef CONTOBANCARIO_H
#define CONTOBANCARIO_H
class ContoBancario{
private:
       char owner[256];
       double balance;
public:
       ContoBancario();
       ContoBancario(char* name, double bal=0);
       ~ContoBancario();
       void Deposit(double amount);
       void Withdraw(double amount);
       double GetBalance();
       void StampaOwner();
};
#endif
```


CLASSI

- La classe `ContoBancario` ha 2 dati membro:

```
char owner[256];
double balance;
```

- I dati membro di una classe sono dichiarati nello stesso modo in cui sono dichiarate le variabili.
- Un dato membro puo' essere di qualsiasi tipo.
- Non e' necessario dichiarare separatamente due membri dello stesso tipo.
- Un dato membro NON puo' essere inizializzato esplicitamente nel corpo della classe. A questo scopo sono utilizzati i *costruttori*.

CLASSI

I dati membro della classe `ContoBancario` sono dichiarati sotto la clausola `private`. Fanno pertanto parte dell'implementazione privata della classe.

```
private:
    char owner[256];
    double balance;
```

Possono far parte dell'implementazione privata anche metodi.

Queste variabili e metodi:

- NON SONO accessibili dall'esterno della classe (ovvero da funzioni esterne alla classe, o dal main)
- SONO accessibili da tutti i metodi della classe (sia pubblici sia privati).

CLASSI - COSTRUTTORI

- sono metodi con lo stesso nome della classe;
- NON possono restituire nessun valore (nemmeno `void`);
- vengono utilizzati per inizializzare i membri della classe;
- vengono chiamati automaticamente quando l'oggetto viene creato, prima che il programmatore possa effettuare qualsiasi operazione sull'oggetto.
- possono essere definiti diversi costruttori, che differiscono per il numero e/o il tipo di parametri (*overloading*):
  - È importante che sia sempre definito un *costruttore di default* (senza argomenti o con argomenti di default per ogni parametro), perché è quello invocato nella creazione degli array:

    ```
    ContoBancario();
    ```
  - possono essere utilizzati argomenti di default;

    ```
    ContoBancario(char* name, double bal=0);
    ```


CLASSI: DISTRUTTORE

- Ha lo stesso nome della classe preceduto da una tilde (~), NON può restituire un valore né ricevere parametri.
- Quando viene distrutto un oggetto viene automaticamente richiamato il relativo distruttore.
- Esegue tutte le operazioni necessarie alla distruzione dell'oggetto, ad esempio:
  - Deallocare la memoria precedentemente allocata dall'oggetto
  - Chiudere files aperti dall'oggetto

CLASSI: OPERATORE ::

- Poiché le definizioni dei metodi sono al di fuori del corpo della classe, è necessario utilizzare un operatore per individuare a quale classe la funzione membro appartenga.

- Tale compito è eseguito dall'operatore *doppi due punti,* `::`, detto operatore del campo d'azione (*scope resolution*). Si utilizza anteponendo ad esso il nome della classe: `nome_classe::`.

CLASSI – DEFINIZIONE DEI METODI

Costruttori della classe `ContoBancario`: *ContoBancario.cpp*

```
#include "ContoBancario.h"
#include <iostream>
using namespace std;
ContoBancario::ContoBancario()
{
        strcpy(owner,"");
        balance=0;
}
ContoBancario::ContoBancario(char *name, double bal)
{
        strcpy(owner,name);
        balance=bal;
}

...
```

File header della classe

Scope resolution

Membri della classe!

CLASSI – DEFINIZIONE DEI METODI

- Il costruttore di default inizializza i membri della classe a valori "di default". In questo caso `owner` e' una stringa vuota e `balance` e' 0.
- Nella classe `ContoBancario` e' definito un secondo costruttore che accetta 2 argomenti:
  `char* name` e `double bal`
- Se il costruttore viene invocato con un solo argomento, verra' utilizzato il valore di default 0 per inizializzare il membro `balance`.
- E' importante notare che nel corpo dei metodi non vengono dichiarate le variabili `owner` e `balance`, in quanto esse sono dati membro della classe.

CLASSI – DEFINIZIONE DEI METODI

Distruttore della classe `ContoBancario`: *ContoBancario.cpp*

```
...
ContoBancario::~ContoBancario()
{
        cout<<"Distruttore! Saldo rimanente: "<<balance<<endl;
}

...
```

- In questo caso non e' necessario deallocare memoria, ne' chiudere files aperti, ma il distruttore, quando richiamato, notifica il saldo rimanente nel `ContoBancario` prima della sua distruzione.
- Noi utilizzeremo il messaggio che viene mandato a monitor per verificare quando il distruttore viene richiamato.

CLASSI – DEFINIZIONE DEI METODI

ContoBancario.cpp

```
...
void ContoBancario::Deposit(double amount)
{
   balance+=amount;
}

void ContoBancario::Withdraw(double amount)
{
   if (amount>balance)
        cout<<"La somma richiesta non e' disponibile!"<<endl;
   else
        balance-=amount;
}
...
```

CLASSI – DEFINIZIONE DEI METODI

- Dato che non e' possibile modificare `balance` dall'esterno della classe, vengono definiti i seguenti metodi per gestire il saldo:

  `void Deposit(double amount)`

  `void Withdraw(double amount)`

- In questo modo (senza un accesso diretto a balance) e' possibile effettuare dei controlli (ad esempio controllarte che `balance` sia maggiore della quantita' che si vuole prelevare, `amount`), in modo tale da non ottenere comportamenti non voluti.

CLASSI – DEFINIZIONE DEI METODI

ContoBancario.cpp

```
...
double ContoBancario::GetBalance()
{
        return balance;
}
void ContoBancario::StampaOwner()
{
        cout<<owner;
}
...
```


CLASSI – DEFINIZIONE DEI METODI

- Per conoscere il valore dei membri `private` dall'esterno della classe si definiscono due metodi:
  - `double GetBalance()`

  (ritorna il valore di balance)
  - `void StampaOwner()`

  (stampa a monitor `owner`)

CLASSI – CREAZIONE DI OGGETTI

- Vediamo ora come puo' essere utilizzata la classe `ContoBancario`:
- Nella funzione `void Test1()` viene creato un oggetto di tipo `ContoBancario`, `conto1`, il cui `owner` e' "Gianni" e con `balance=123.45`.
- Si testano i metodi della classe per effettuare depositi e prelievi e si stampa a monitor il suo stato, prima e dopo ogni operazione.

CLASSI – CREAZIONE DI OGGETTI

test.cpp

```
...
void Test1(){
   ContoBancario conto1("Gianni",123.45);
   //cout<<conto1.balance;
   cout<<"Saldo attuale:"<<conto1.GetBalance()<<endl;
   conto1.Deposit(100.50);
   cout<<"Saldo attuale:"<<conto1.GetBalance()<<endl;
   conto1.Withdraw(99.10);
   cout<<"Saldo attuale:"<<conto1.GetBalance()<<endl;
   conto1.Withdraw(300.50);
   cout<<"Saldo attuale:"<<conto1.GetBalance()<<endl;
}
...
```

costruttore

Errore compile-time `balance` e' private

Una possibile uscita:

```
Saldo attuale:123.45
Saldo attuale:223.95
Saldo attuale:124.85
La somma richiesta non e' disponibile!
Saldo attuale:124.85124.85
Distruttore! Saldo rimanente:
```

Chiamata al distruttore

CLASSI – CREAZIONE DI OGGETTI

- Nella funzione `void Test2()` vengono creati due oggetti di tipo `ContoBancario`, `conto1` e `conto2`, i cui owner sono rispettivamente "Gianni" e "Maria" e i `balance 123.45 e 78.90.`
- Si testano i metodi della classe per i due diversi oggetti.

CLASSI – CREAZIONE DI OGGETTI

test.cpp

```
...
void Test2()
{
       ContoBancario conto1("Gianni",123.45);
       ContoBancario conto2("Maria",78.90);
       cout<<"Saldo attuale di ";
       conto1.StampaOwner();
       cout<<": "<<conto1.GetBalance()<<endl;
       cout<<"Saldo attuale di ";
       conto2.StampaOwner();
       cout<<": "<<conto2.GetBalance()<<endl;
}
...
```

Una possibile uscita:

```
Saldo attuale di Gianni: 123.45
Saldo attuale di Maria: 78.9
Distruttore! Saldo rimanente: 78.9
Distruttore! Saldo rimanente: 123.45
```

Chiamate al distruttore

CLASSI – CREAZIONE DI OGGETTI

- Nella funzione `void Test3()` viene creato un oggetto di tipo `ContoBancario`, `contodef` utilizzando il costruttore di default.
- Si testano i metodi della classe per verificare il valore di owner e balance.

CLASSI – CREAZIONE DI OGGETTI

test.cpp

```
...
void Test3()
{
        ContoBancario contodef;
        cout<<"Saldo attuale di ";
        contodef.StampaOwner();
        cout<<": "<<contodef.GetBalance()<<endl;
}
...
```

Costruttore di default

Una possibile uscita:

```
Saldo attuale di : 0
Distruttore! Saldo rimanente: 0
```