Le Classi in C++

Sommario

- Le strutture
- Le classi

Introduzione

- Iniziamo a parlare di strutture dati di tipo *struct* cioè di aggregati di dati
- Muoveremo poi dalla analisi dei limiti di queste strutture per introdurre il concetto di *classe*
- Definiremo in modo preciso la sintassi messa a disposizione dal C++ per istanziare e manipolare oggetti

Le strutture

- Le strutture sono insiemi (aggregati) di tipi di dati diversi
- Es:

```
struct Time{
   int hour;
   int minute;
   int second;
};
```

- La parola chiave struct introduce la definizione
- I tipi nel corpo della struttura sono chiamati membri della struttura

Caratteristiche delle strutture

- Le strutture:
  - **possono** contenere qualsiasi tipo di dato
  - **possono** contenere altre strutture
  - **non possono** contenere istanze di se stesse
  - **possono** contenere puntatori al proprio tipo di struttura e vengono dette strutture auto-referenti (saranno utili per formare strutture dati come *liste* e *alberi*)

- **NOTA:** La dichiarazione di una struttura non alloca spazio in memoria ma dichiara solo un tipo nuovo

Uso delle strutture

- Una struttura si può poi usare come un tipo:
  - Time timeObj;
  - Time timeArray[100];
  - Time *timePtr;
  - Time &timeRef=timeObj;

Accesso ai membri di una struttura

- Si fa accesso ai membri di una struttura tramite l'operatore di accesso
  - Se la struttura e' memorizzata sullo stack (memoria statica) si usa "**.**"
  - Se la struttura e' memorizzata sullo heap (memoria dinamica) si usa "**->**"

Esempio

```
Time timeObj;
Time timeArray[100];
Time *timePtr;
Time &timeRef=timeObj;

cout<< timeObj.hour;
cout<<timeRef.hour;
timePtr=&timeObj;

cout<<timePtr->hour;
cout<< (*timePtr).hour;
```

- **Nota:** è un errore non mettere le parentesi a causa della precedenza dell'operatore . rispetto a *
- Senza parentesi si avrebbe:

```
*timePtr.hour = *(timePtr.hour)
```

Esempio

```
struct Time {     // structure definition
   int hour;      // 0-23
   int minute;    // 0-59
   int second;    // 0-59
};

void printMilitary( const Time & );  // prototype
void printStandard( const Time & );  // prototype

int main()
{
   Time dinnerTime;    // variable of new type Time

   // set members to valid values
   dinnerTime.hour = 18;
   dinnerTime.minute = 30;
   dinnerTime.second = 0;
```

Esempio

```
   cout << "Dinner will be held at ";
    printMilitary( dinnerTime );
    cout << " military time,\nwhich is ";
    printStandard( dinnerTime );
    cout << " standard time.\n";

    //set members to invalid values
    dinnerTime.hour = 29;
    dinnerTime.minute = 73;

    cout << "\nTime with invalid values: ";
    printMilitary( dinnerTime );
    cout << endl;
    return 0;
}
```

Esempio

```
// Print the time in military format
void printMilitary( const Time &t )
{
   cout << ( t.hour < 10 ? "0" : "" ) << t.hour << ":"
        << ( t.minute < 10 ? "0" : "" ) << t.minute;
}

// Print the time in standard format
void printStandard( const Time &t )
{
   cout << ( ( t.hour == 0 || t.hour == 12 ) ?
               12 : t.hour % 12 )
        << ":" << ( t.minute < 10 ? "0" : "" ) << t.minute
        << ":" << ( t.second < 10 ? "0" : "" ) << t.second
        << ( t.hour < 12 ? " AM" : " PM" );
}
```

Inconvenienti con le strutture

- Non esiste inizializzazione: pertanto si possono avere membri non inizializzati
- Non esiste controllo sui valori assegnati ai membri: è possibile che vengano assegnati valori fuori dagli intervalli concettualmente corretti (es. un valore per l'ora > 24)
- Se si cambia l'implementazione della struttura (ad es si potrebbero memorizzare i secondi dalla mezzanotte) occorre cambiare qualcosa in tutte le parti del programma in cui si è utilizzata la struttura

Inconvenienti

- Non si possono trattare le strutture come come oggetti atomici:
  - Non si possono confrontare direttamente per vedere se due istanze sono uguali (ma si devono confrontare singolarmente tutti i membri)
  - Non si può stampare una struttura (ma si deve farlo per ogni sua parte)

...e adesso le classi

- Di seguito vediamo come trattare lo stesso problema attraverso il concetto di classe
- Nelle lezioni seguenti vedremo i dettagli di come inizializzare gli oggetti, deallocarli, etc

Esempio

```
// Time abstract data type (ADT) definition
class Time {
public:
   Time();                              //constructor
   void setTime( int, int, int );       //set hour, minute, second
   void printMilitary();                //print military time format
   void printStandard();                //print standard time format
private:
   int hour;      // 0 - 23
   int minute;    // 0 - 59
   int second;    // 0 - 59
};
```

Esempio

```
// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}
```

Esempio

```
// Print Time in military format
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Print Time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Esempio

```
int main()
{
   Time t;  // instantiate object t of class Time

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime( 13, 27, 6 );
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();
   t.setTime( 99, 99, 99 );  // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:"
        << "\nMilitary time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

Il concetto di classe

Programmazione orientata agli oggetti (Object Oriented Programming OOP):

- si incapsulano i dati e le funzioni all'interno di classi
  - le funzioni sono strettamente correlate con i dati che manipolano
- le classi definiscono dei prototipi per gli oggetti
  - sono una specie di progetto che può essere utilizzato per realizzare tanti oggetti dello stesso tipo
- un oggetto è una istanza di una classe
  - fra una classe e un oggetto vi è la stessa relazione che sussiste fra un tipo e una variabile

Il concetto di classe

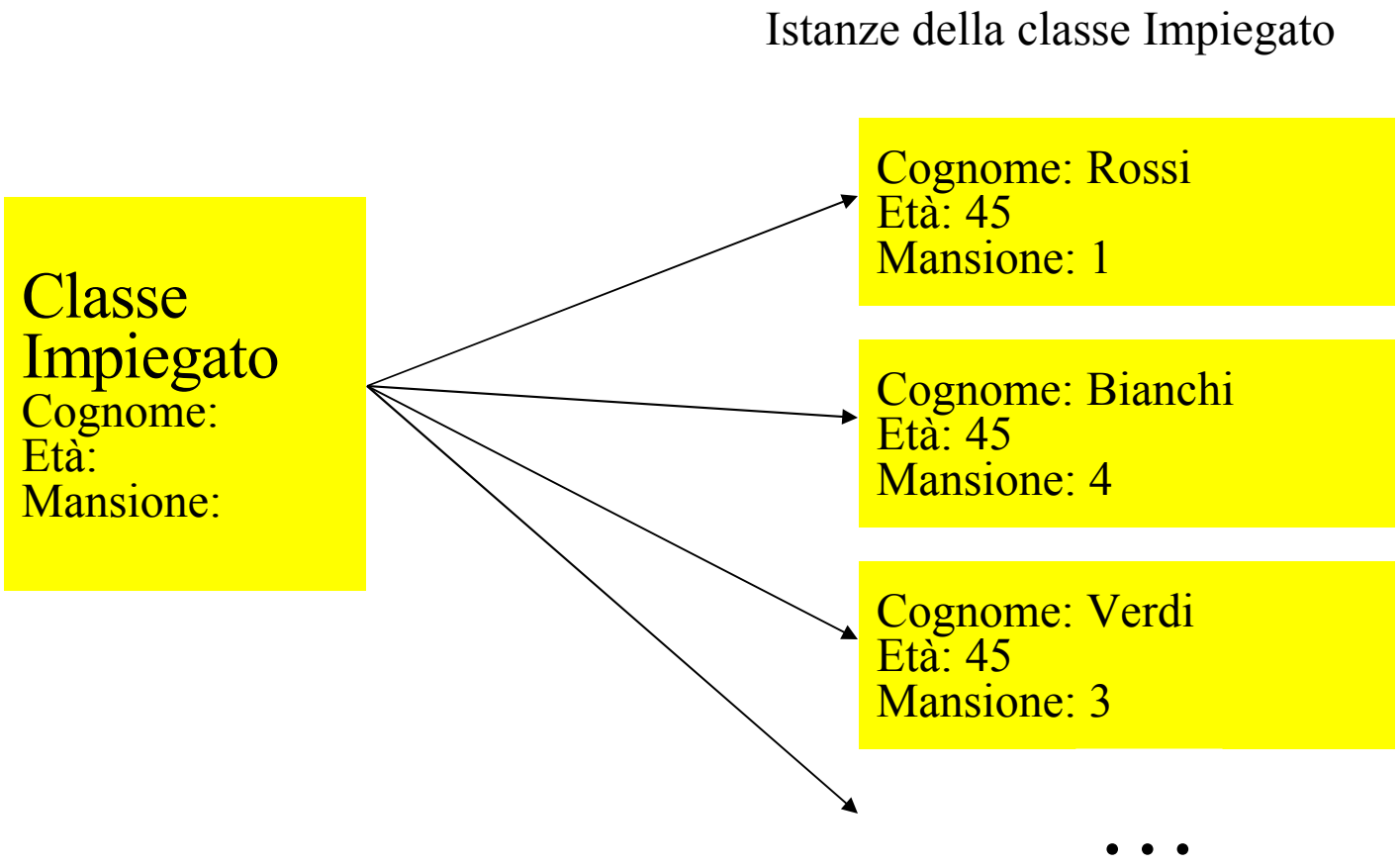
- Proprietà fondamentale è l'occultamento delle informazioni:
  - i dettagli rimangono nascosti e non visibili al di fuori della classe
  - questo è un argomento fondamentale per garantire robustezza da un punto di vista dell'ingegneria del software
  - è come guidare un automobile senza conoscere il funzionamento del motore, la trasmissione e il sistema di alimentazione del carburante: non solo è possibile, ma la maggiore astrazione permette di guidare auto diverse con la stessa facilità

Il concetto di classe

- In C la programmazione tende ad essere orientata all'azione
- l'unità fondamentale di programmazione è la funzione
- chi programma in C andrà alla ricerca dei *verbi* nella specifica di un sistema

- In C++ la programmazione è orientata sulla definizione dei tipi definiti dagli utenti
- l'unità fondamentale di programmazione sono gli oggetti
- chi programma in C andrà alla ricerca dei *nomi* nella specifica di un sistema

Il concetto di classe



Il concetto di classe

Una classe è costituita da:

- **Dati:** dati membro o attributi
- **Funzioni:** funzioni membro o metodi

Esempio:

```
class Impiegato{
    public:
    Impiegato();
    ~Impiegato();
    void assegna_nome_cognome(char*, char*);
    void assegna_reparto(int);
    const char* recupera_nome();
    const char* recupera_cognome();
    const int recupera_reparto();
    void stampa();
    private:
    char* nome;
    char* cognome;
    int id;
    int id_reparto;
};
```

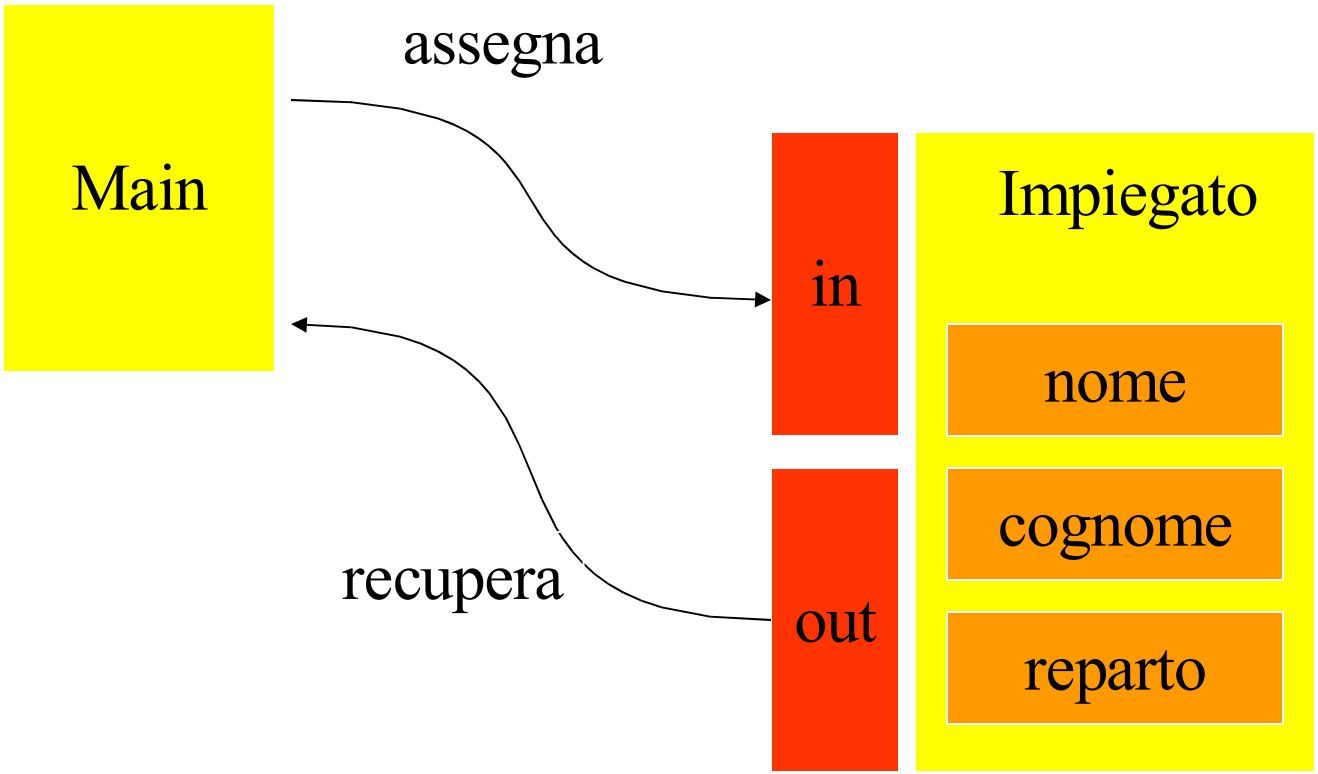
Funzioni membro

Dati membro

Information Hiding

- Il principio di information hiding generalizza il concetto di ADT per qualsiasi aspetto di un programma
  - Gli oggetti comunicano attraverso interfacce
  - Non è nota l'implementazione della classe all'esterno della classe stessa
  - Questo permette la riusabilità e la manutenzione

Funzioni di interfaccia



Anatomia di una classe

- Dichiarazione e definizione di una classe

```
class NomeClasse {
public:
    void set(int); //membri pubblici
private:
    int a; //membri privati
};

void NomeClasse::set(int var) {
    a=var;
}
```

Dichiarazione

- Una classe si dichiara con la parola chiave class
- La dichiarazione del tipo dei suoi membri è fra “{ }” e terminata da “;”
- I membri possono essere in una fra le seguenti sezioni:
  - public
  - protected
  - private
- Il default per le classi è *private* (vedremo dopo)

Definizione di una funzione membro

- Le funzioni membro di una classe possono essere definite:
  - all'interno della dichiarazione di classe
  - all'esterno tramite l'operatore "::" detto operatore binario di risoluzione di visibilità

```
tipo_rest NomeClasse::funzione(tipo parametro){

    //definizione della funzione
    //....
}
```

Il qualificatore `inline`

- Il qualificatore **`inline`** suggerisce al compilatore di copiare il codice della funzione nel punto di utilizzo invece di eseguire una chiamata a funzione (più costosa)

- Le funzioni definite all'interno della dichiarazione di classe sono rese automaticamente **`inline`**

Esempio:

```
Class Dato{
   public:
        Dato();
        void set(int a){num=a;} //DICHIARAZIONE inline automatica
        int get();
        int do();
   private:
        int num;
};

int Dato::get(){return num;} //DICHIARAZIONE non inline
inline int Dato::get(){return num;} //DICHIARAZIONE inline
```