I membri e le restrizioni di accesso

Sommario

- I membri
- Le restrizioni di accesso
- Separazione interfaccia da implementazione

Accedere ai membri di una classe

- La visibilità dall'esterno dei dati e delle funzioni membro di una classe dipende dalle restrizioni imposte
- All'interno di una classe tutti i membri sono visibili e accessibili tramite il proprio nome
- Le variabili definite all'interno di una funzione membro sono visibili solo alla funzione stessa
  - In caso di nomi coincidenti fra variabili locali ad una funzione membro e dati membro si deve ricorrere all'operatore "::" per accedere al dato membro
- Per accedere dall'esterno si usano gli stessi operatori di accesso usati per le struct "." e "**->**"

Accedrere ai dati membro

```
// Simple class Count
class Count {
public:
   int x;
   void print() { cout << x << endl; }
};

int main(){
   Count counter,                  // create counter object
         *counterPtr = &counter, // pointer to counter
         &counterRef = counter;  // reference to counter

   cout << "Assign 7 to x and print using the object's name: ";
   counter.x = 7;          // assign 7 to data member x
   counter.print();        // call member function print
   cout << "Assign 8 to x and print using a reference: ";
   counterRef.x = 8;       // assign 8 to data member x
   counterRef.print();     // call member function print

   cout << "Assign 10 to x and print using a pointer: ";
   counterPtr->x = 10;   // assign 10 to data member x
   counterPtr->print(); // call member function print
   return 0;
}
```

Il controllo dell'accesso

- E' possibile controllare l'accesso che viene consentito ai dati ed alle funzioni membro di una classe
- I possibili tipi di controllo sono:
  - pubblico
  - privato
  - protetto (vedremo in seguito)
- Tutti i dati e le funzioni dichiarate dopo lo specificatore *public:* sono ad accesso pubblico
- Tutti i dati e le funzioni dichiarate dopo lo specificatore *private:* sono ad accesso privato

Accesso di tipo *public*

- Lo scopo dei membri *public* è di presentare al client di una classe una interfaccia di servizi disponibili
- I dati e le funzioni *public* sono accessibili a tutte le altre funzioni del programma

Accesso di tipo *private*

- I membri *private* sono accessibili solo alle funzioni membro di tale classe
- Non è possibile per alcuna funzione esterna accedere ai dati o funzioni *private* di una classe

Sintassi

- La modalità di accesso di default è *private*
- Tutti i membri elencati nell'intestazione di una classe dall'inizio al primo specificatore esplicito sono considerati private
- Dopo ogni specificatore la modalità indicata si applica a tutti i membri fino allo specificatore successivo o al termine della definizione della classe

Restrizione di accesso

- Vale sempre il principio del minimo privilegio:

  Consentire il livello necessario di accesso e niente di più

- Nella maggior parte dei casi:
  - i dati membro dovrebbero essere privati
  - le funzioni membro dovrebbero essere pubbliche
  - funzioni membro di servizio (utilizzate da altre funzioni membro pubbliche) dovrebbero essere private

Restrizioni default

- In una struct la modalità di accesso di default è *public*
- Una struct in C++ è una classe che *logicamente* non dovrebbe possedere funzioni membro, ma sintatticamente non ci sono limitazioni
- In realta' una struct o una classe in C++ sono equivalenti a meno della modalita' di accesso di default

Funzioni di utilità

- Non tutte le funzioni membro devono essere pubbliche per essere utili
- Se una funzione esegue dei compiti necessari ad altre funzioni membro allora può essere resa private ed essere unicamente utilizzata all'interno delle altre funzioni
- Ad esempio: le *funzioni predicative* come isEmpty() o isValid() che controllano la verità o falsità di una condizione

Esempio di funzione di utilità:

```
class Tempo{
   public:
        Tempo();
        void set(int, int);
        void stampa();
        void stampa_secondi();
   private:
        int ora;
        int min;

        int converti_sec();
};

void Tempo::stampa_secondi(){
   int sec=converti_sec();
   cout<<"Secondi da mezzanotte: "<<sec<<endl;
}

int Tempo::converti_sec(){
    return min*60+ora*3600;
}
```

E' usata in una funzione membro ma non serve renderla accessibile all'esterno

Separazione interfaccia implementazione

- Un principio fondamentale di buona ingegneria del software è separare *l'interfaccia dall'implementazione*
- Per ottenere:
  - modifiche più facili
  - cambiamenti all'implementazione trasparenti al client
- Tutto quello che deve sapere un programma client è contenuto nelle dichiarazioni public di una classe;
- L'implementazione delle funzioni e tutto ciò che è private può essere cambiato in modo trasparente senza dover effettuare alcuna modifica sul programma client

Divisione in più file

- La dichiarazione di una classe va in un header file che il software client include se vuole usare la classe (ex. Tempo.h)
- Le definizioni delle funzioni membro vanno in un file sorgente che viene collegato in fase di link dal compilatore (ex. Tempo.cpp)
- In questo modo i venditori di software possono rilasciare gli header con le interfaccie e le definizioni in forma di programmi oggetto

Il problema delle dichiarazioni multiple

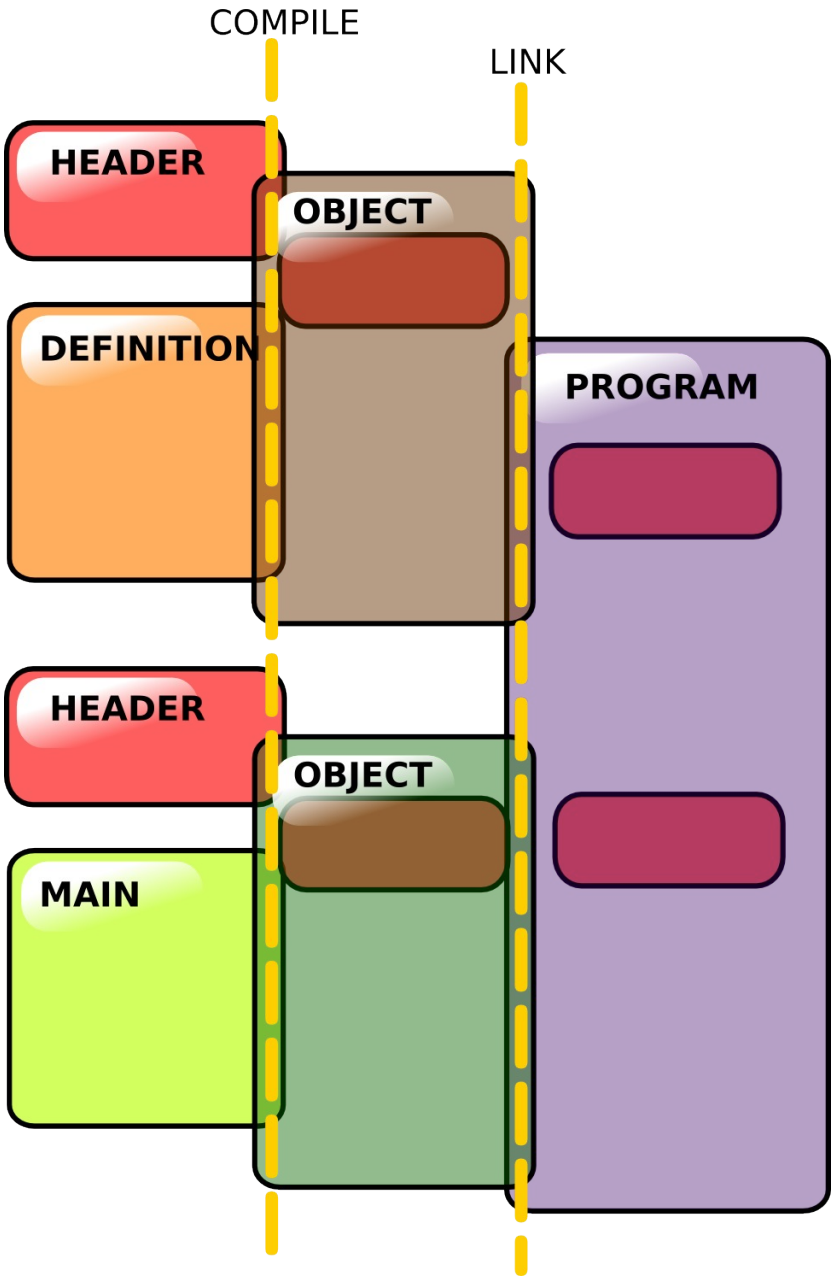
- Se si vuole compilare un programma sorgente si deve fornire la dichiarazione delle variabili, funzioni e classi e la loro definizione
- Se si vuole utilizzare una variabile, funzione o classe definita altrove in un programma e' sufficiente che il programma ne conosca la dichiarazione
- Successivamente in fase di link verra' collegata la definizione vera e propria
- ...ma quando si collegano i due compilati (il programma e la definizione della funzione) si hanno due dichiarazioni
- Non e' mai possibile avere due definizioni della stessa entita' (perche' non si riesce a decidere quale usare)

L'inclusione condizionale

- La direttiva di preprocessore #ifndef e #define permettono di evitare l'inclusione ripetuta dello stesso file
- Usando *#ifndef var* si accede alla parte rimanente solo nel caso in cui non sia mai stata definita *var*
- Ma l'istruzione successiva definisce proprio *var*
- In questo modo la prima volta che si incontra #ifndef si includera' la definizione mentre la seconda volta no

- Si dovrebbe sempre utilizzare questa tecnica
- Si consiglia di definire costanti dal nome mutuato dal file a cui si riferiscono
  - Es. TIME_H da Time.h

Inclusione Condizionale



Esempio: File time1.h

```
#ifndef TIME1_H
#define TIME1_H
// Time abstract data type 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
};
#endif
```

Esempio: File time1.cc

```
// Member function definitions for Time class.
#include <iostream>
#include "time1.h"

// 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: File time1.cc

```
// 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: File main.cc

```
#include <iostream>
#include "time1.h"

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:\n"
        << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

Esempio: File salesp.h

```
// SalesPerson class definition
// Member functions defined in salesp.cpp
#ifndef SALESP_H
#define SALESP_H

class SalesPerson {
public:
   SalesPerson();                    // constructor
   void getSalesFromUser(); // get sales figures from keyboard
   void setSales( int, double ); // User supplies one month's
                                     // sales figures.
   void printAnnualSales();
private:
   double totalAnnualSales();      // utility function
   double sales[ 12 ];             // 12 monthly sales figures
};
#endif
```

Esempio: File salesp.cc

```
#include "salesp.h"
// Member functions for class SalesPerson
// Constructor function initializes array
SalesPerson::SalesPerson()
{
   for ( int i = 0; i < 12; i++ )
      sales[ i ] = 0.0;
}

//Function to get 12 sales figures from the user at the keyboard
void SalesPerson::getSalesFromUser()
{
   double salesFigure;
   for ( int i = 1; i <= 12; i++ ) {
      cout << "Enter sales amount for month " << i << ": ";
      cin >> salesFigure;
      setSales( i, salesFigure );
   }
}
```

Esempio: File salesp.cc

```
// Function to set one of the 12 monthly sales figures.
// Note that the month value must be from 0 to 11.
void SalesPerson::setSales( int month, double amount ){
   if ( month >= 1 && month <= 12 && amount > 0 )
      sales[month-1]=amount;//adjust for subscripts 0-11
   else
      cout << "Invalid month or sales figure" << endl;
}
// Print the total annual sales
void SalesPerson::printAnnualSales(){
   cout << setprecision( 2 )
        << setiosflags( ios::fixed | ios::showpoint )
        << "\nThe total annual sales are: $"
        << totalAnnualSales() << endl;
}
// Private utility function to total annual sales
double SalesPerson::totalAnnualSales(){
   double total = 0.0;
   for ( int i = 0; i < 12; i++ )
      total += sales[ i ];
   return total;
}
```

Esempio: File main.cc

```
#include "salesp.h"

int main()
{
   SalesPerson s;           // create SalesPerson object s
   s.getSalesFromUser();    // note simple sequential code
   s.printAnnualSales();    // no control structures in main
   return 0;
}
```