Classi

C e C++

- Fino a questo punto abbiamo introdotto gli elementi base di qualsiasi linguaggio di programmazione, ed in particolare quelli comuni al C e al C++.
- Tutto ciò che abbiamo visto è valido anche per il C con pochi semplici accorgimenti
  - Sostituzione di iostream.h con stdio.h (e degli oggetti cin e cout con le funzioni scanf e printf)
  - Sostituzione di new e delete con malloc o calloc e free

Oggetti

- Abbiamo introdotto diversi oggetti: oggetti numerici (int n, double x, char lettera...), vettori di oggetti numerici tutti dello stesso tipo (double dati[10], string nome o char nome[20]...), ma anche oggetti di un nuovo tipo: cin (tipo istream) e cout (tipo ostream)...

Tipi

- Oltre ai tipi predefiniti (int, float, double, char, long, short, unsigned int...) che corrispondono a singoli oggetti numerici sia il C che il C++ consentono di definire dei nuovi tipi associati a degli insiemi di oggetti, anche di tipo non omogeneo.
- Esempio: documento di identità, definiamo il tipo CI

```
int numero_documento;
string nome;
string cognome;
string luogo_nascita;
float altezza;
int data_nascita; // nel formato ggmmaa
```

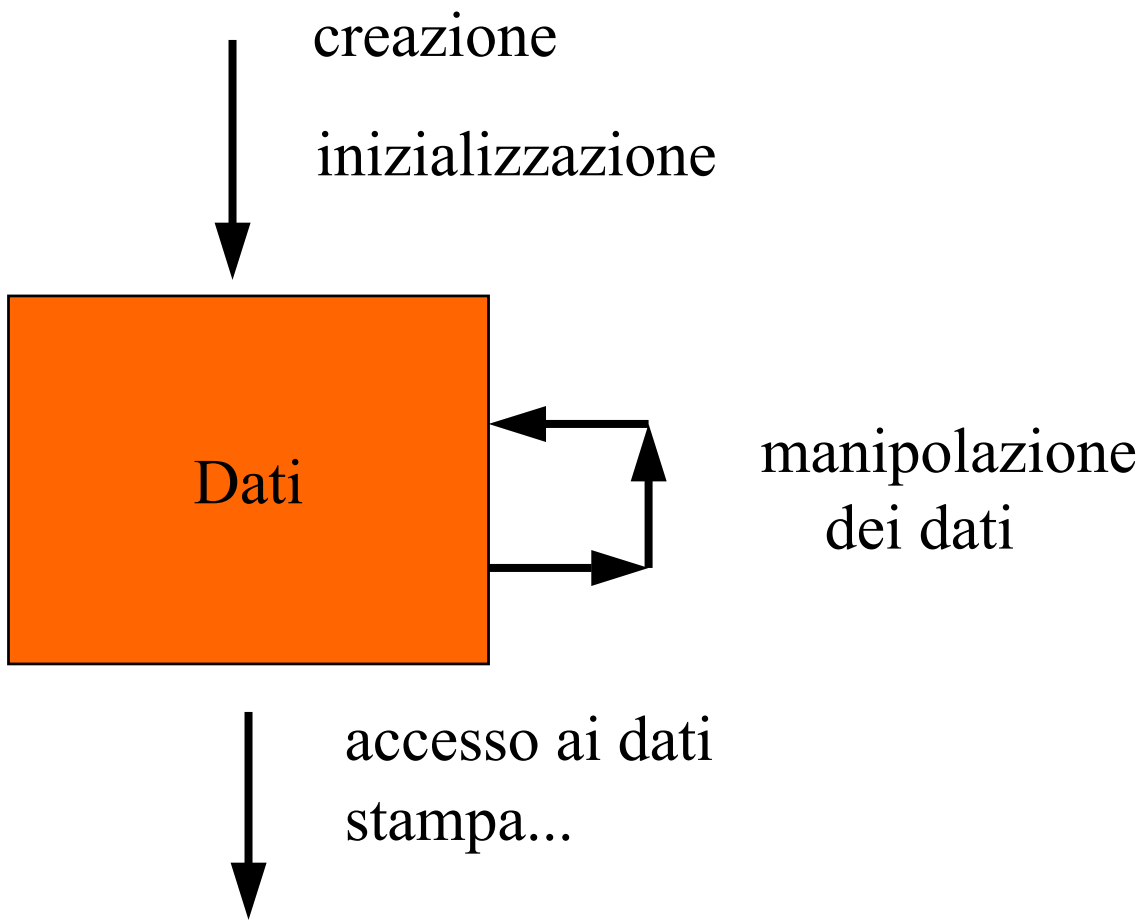
Nuovi tipi

- In C si possono definire degli oggetti composti associati a dei nuovi tipi mediante dei costrutti chiamati strutture (struct) o unioni (union).
- In C++ si possono ancora usare le strutture del C ma si dispone di cotrutti molto più potenti: le classi.

Classe

- Una classe è un nuovo tipo di oggetto (come int, float, double...) con la possibilità:
  - Di far corrispondere all'oggetto un insieme di valori numerici anche di tipo non omogeneo
  - Di decidere se (e come) tali valori possano essere acceduti dall'esterno
  - Di definire delle funzioni responsabili della manipolazione di questi dati (metodi)

Oggetti di una classe



Uso di una classe predefinita

- Come nel caso delle funzioni di libreria esistono delle classi predefinite che il programmatore può utilizzare (istream e ostream per esempio).
- Per usarle è sufficiente conoscere la loro interfaccia (e includerla nel proprio codice), ovvero di che tipo di oggetti si tratta e quali siano i metodi attraverso i quali gli oggetti possono essere acceduti dall'esterno (creati, inizializzati, cancellati, stampati, usati...).
- Questi metodi sono metodi pubblici, chiamabili da tutti, altri, dichiarati privati, possono essere chiamati solo da altri metodi della classe stessa.

Sintassi

```
class identificatore {
   accesso1:
        membro11...
   ...
   accessoN:
        membro N1…
        membro N2
   };
```

identificatore è un qualsiasi identificatore (nome) consentito

da accesso1 fino a accessoN possono essere:

public, private o protected.

Ogni membro è o un dato o una funzione (metodo).

Un membro dato è una dichiarazione di un oggetto.

Una membro funzione è la dichiarazione di una funzione o la sua implementazione.

Esempio

- Costruiamo un nuovo tipo, **Punto2D**, che descriva un punto nel piano. Le sue proprietà sono:
  - Dati privati: 2 coordinate
  - Metodi:
    - Creazione di un oggetto punto date le 2 coordinate,
    - Accesso in lettura alle 2 coordinate,
    - Calcolo della distanza tra due punti
    - Stampa dei parametri

Punto2D.h (dichiarazione)

```
#ifndef Punto2D_H
#define Punto2D_H

   class Punto2D {

   private:
          double coordX;     // coordinata X
          double coordY;     // coordinata Y

   public:
          Punto2D(  );       // crea senza inizializzare
          Punto2D(double X, double Y); // crea da(X,Y)
          double distanza(Punto2D punto); // distanza
          double x(  );              // restituisce X
          double y(  );              // restituisce Y
          void stampa(  );  // stampa (X,Y)
   };
   #endif
```

Punto2D.cc (implementazione)

```
#include "Punto2D.h"
#include <iostream.h>
#include <math.h>

Punto2D::Punto2D(  ) { }

Punto2D::Punto2D(double X, double Y) {
    coordX=X;
    coordY=Y;
}
```

```
double Punto2D::distanza(Punto2D punto) {
    double deltaX = coordX - punto.x(  );
    double deltaY = coordY - punto.y(  );
          return sqrt( deltaX*deltaX +
deltaY*deltaY );
}

double Punto2D::x(  ) {return coordX; }

double Punto2D::y(  ) {return coordY;}

void Punto2D::stampa(  ) {
    cout << "(" << coordX << ","
         << coordY << ")" << endl;
}
```

Uso di Punto2D

```
#include <iostream.h>
#include "Punto2D.h"
   int main(  ) {
       Punto2D origine(0,0);
       origine.stampa(  ); // oggetto.metodo( )
       double x,y;
       cin >> x >> y;
       Punto2D punto(x,y);
       punto.stampa(  );
       Punto2D * puntatore;
       puntatore = &punto;
       puntatore->stampa(  ); // puntatore->metodo( )
       return 1;
   }
```

Esempio

- Costruiamo un nuovo tipo, **Retta2D**, che descriva una retta nel piano. Le sue proprietà sono:
  - Dati privati: 3 parametri
  - Metodi:
    - Creazione dati 2 punti
    - Creazione dati 1 punto e una direzione
    - Calcolo dell'intersezione con un'altra retta
    - Calcolo della distanza da un punto
    - Accesso ai 3 parametri e stampa dei parametri

  (se ne potrebbero pensare tanti altri: creazione dati i 3 parametri, ricerca di una retta ortogonale etc...)

Retta2D.h (dichiarazione)

```
#ifndef Retta2D_H
#define Retta2D_H
#include "Punto2D.h"

class Retta2D {

private:
          // Ax + By + C = 0
          double A;
          double B;
          double C;

public:

     double a(  );      // restituisce A
     double b(  );      // restituisce B
     double c(  );      // restituisce C
     void stampa(  );// stampa A,B,C
```

```
public:
   Retta2D(Punto2D p1, Punto2D p2);
   // crea la retta passante per p1 e p2

   Retta2D(Punto2D p, double xDir, double yDir);
   // crea la retta per p parallela al vettore
   //(xDir,yDir)

   Punto2D intercetta(Retta2D altra_retta);
   // punto di intersezion con altra_retta

   double distanza(Punto2D punto);
   // distanza da un punto

   static double toll_parallelismo;
   // angolo minimo (rad) tra rette non parallele
};
#endif
```

Retta2D.cc: implementazione

```
#include "Retta2D.h"
#include <iostream.h>
#include <math.h>

double Retta2D::toll_parallelismo = 0.01;
// questo è un dato pubblico della classe, condiviso da
// tutti gli oggetti Retta2D. Averlo definito con la
// parola chiave static permette di riservargli la memoria
// una sola volta anzichè farlo per ogni oggetto
// Retta2D creato
```

```
Retta2D::Retta2D(Punto2D p1, Punto2D p2) {
      if( p1.x() == p2.x() ) {
      // retta verticale
            A = 1;
            B = 0;
            C = -p1.x();
      } else {
            A = p2.y() - p1.y();
            B = p1.x() - p2.x();
            C = p1.y()*p2.x() - p2.y()*p1.x();
      }
   }

Retta2D::Retta2D(Punto2D p, double xDir, double
   yDir) {
      A = yDir;
      B = -xDir;
      C = - A*p.x() - B*p.y();
   }
```

```
Punto2D Retta2D::intercetta(Retta2D altra_retta)
   {
   double det = A * altra_retta.b(  ) -
   altra_retta.a(  ) * B;
   double sin2 = ( det*det ) / ( (A*A + B*B)*
              (altra_retta.a(  )*altra_retta.a(
   )+
              altra_retta.b(  )*altra_retta.b(  ))
   );
   if( sin2 <
   toll_parallelismo*toll_parallelismo) {
       return Punto2D(FLT_MAX,FLT_MAX);
   } else {
       double X = (B*altra_retta.c(  ) -
                   altra_retta.b(  )*C)/det;
       double Y = (C*altra_retta.a(  ) -
                   altra_retta.c(  )*A)/det;
       return Punto2D(X,Y);
   }
}
```

```
double Retta2D::distanza(Punto2D punto) {
   return
   fabs( A*punto.x(  ) + B*punto.y(  ) +
   C)/sqrt(A*A+B*B);
}

double Retta2D::a(  ) { return A; }

double Retta2D::b(  ) { return B; }

double Retta2D::c(  ) { return C; }

void Retta2D::stampa(  ) {
   cout << A << " x + " << B << " y + " << C << " =0"
        << endl;
}
```

Modifiche in Punto2D.h

- Aggiungiamo un metodo per calcolare la distanza tra il Punto2D ed una retta di tipo Retta2D tra i metodi public

  double distanza(Retta2D retta);

- Per far conoscere la classe Retta2D a Punto2D dovremmo includere Retta2D.h, ma Retta2D.h include a sua volta Punto2D.h. Per evitare problemi ci limitiamo quindi a dire in Punto2D.h, prima della dichiarazione di class Punto2D, che esiste una classe Retta2D, mediante la dichiarazione

  class Retta2D;

Modifiche in Punto2D.cc

- Aggiungiamo l'implementazione del metodo distanza, che utilizza il metodo della classe Retta2D

```
double Punto2D::distanza(Retta2D retta) {
   return retta.distanza(*this);
  // this è il puntatore all'oggetto
   // stesso e *this è
  // l'oggetto stesso (Punto2D)
}
```

- Adesso non basta più sapere che esiste una classe Retta2D: usiamo un suo metodo, ci serve la sua dichiarazione! Punto2D.cc deve quindi includere Retta2D.h

```
#include "Retta2D.h"
```

Uso di Retta2D e Punto2D

```
#include <iostream.h>
#include "Punto2D.h"
#include "Retta2D.h"

int main(  ) {
    Punto2D origine(0,0);
    Punto2D P(1,1);

    Retta2D R(origine,P);
    R.stampa(  );

    return 1;
}
```

Altri esempi

- timeObject
- Date

Esempio timeObject: dichiarazione (timeObject.h)

```
class timeObject{
  private:
      int hours;
      int minutes;
      int seconds;
  public:
      void setTime(int, int, int); //impostazione
      void increment(int,int,int); //variazione
      void display(  );// stampa
      int after(timeObject); // confronto
      int operator==(timeObject); //confronto
};// attenzione, il ; finale serve!
```

timeObject:implementazione
(timeObject.cc)

```
void timeObject::setTime(int h,int m,int s){
      hours = h;
      minutes = m;
      seconds = s;
}
void timeObject::increment(int h,int m,int s){
      hours = hours + h;
      minutes = minutes + m;
      seconds = seconds + s;
}
void timeObject::display(  ){
      cout << hours <<" hours: "
           << minutes << " minutes: "
           << seconds << " seconds: "
                      << endl;
}
```

```
int timeObject::after(timeObject t){
      if (hours > t.hours)return 1;
      if(hours < t.hours) return 0;
      if (minutes > t.minutes)return 1;
      if(minutes < t.minutes) return 0;
      if (seconds > t.seconds)return 1;
      if(seconds < t.seconds) return 0;
      return 0;
   }

int  timeObject::operator==(timeObject t){
      return (hours == t.hours &&
                  minutes == t.minutes &&
                  seconds == t.seconds);
   }
```

Esempio: programma che utilizzi timeObject

```
#include  <iostream.h>
#include  "timeObject.h"
int main(  ){
   timeObject t;          // dichiarazione
   t.setTime(10,4,21);    //inizializzazione
   t.display(  );              // stampa
   t.increment(1,5,31)    // variazione
   t.display(  );              // stampa
   timeObject newTime;    // dichiarazione
   newTime.setTime(11,15,45);//inizializzazione
   cout<<(t == newTime)<<endl; // confronto
   return 1;}
```

Esempio Date: dichiarazione (Date.h)

```
class Date {
  private:
       int day;
       int month;
       int year;
  public:
       Date(int=1,int=1,int=0);   // costruttore
       void setDay(int d){ day = d;}// impostazione day
       void setMonth(int m){month =m;}// imp. month
       void setYear(int y){year =y;}// imp. year
       int  getDay(  ){return day;} // lettura day
       int  getMonth(  ){return month;} // lettura month
       int  getYear(  ){return year;} // lettura year
       void usDisplay(  );
       void interDisplay(  );
};
```

Date: implementazione (Date.cc)

```
Date::Date(int d, int m, int y){
          day = d;
          month = m;
          year = y;
}

void Date::usDisplay( ){
    cout<<month<<"/";
    cout<<day<<"/"<<year;
}

void Date::interDisplay( ){
    cout<<day<<"/";
    cout<<month<<"/"<<year;
}
```

Metodi indispensabili

- Per ogni classe sono indispensabili i seguenti metodi:
  - Costruttori
  - Distruttore
- Il compilatore definisce automaticamente il costruttore di un oggetto non inizializzato e quindi se non volete aggiungere qualcosa in questo metodo (ad esempio una stampa che dica che viene creato un nuovo oggetto) non serve dichiararlo ed implementarlo. Analogamente il compilatore definisce automaticamente il costruttore di copia.

Costruttore di copia

Sintassi:

idClasse(const idClasse &);

Il costruttore di copia viene invocato in tre situazioni:

1 dichiarazioni con assegnazione:

```
Punto2D p(0,0);
Punto2D pbis=p;
```

2 quando si passi un oggetto come parametro in una chiamata a una funzione:

```
retta.distanza(punto);
```

3 quando un oggetto viene ritornato da una funzione

```
punto=retta.intersezione(altra_retta);
```

Distruttore

sintassi:

```
~idClasse(  );
```

Il distruttore libera la memoria occupata dall'oggetto.

Il distruttore viene chiamato dal compilatore quando si esce dallo *scope* in cui è stato definito l'oggetto:

```
{
    Punto2D p(0,0);
}   // viene chiamato ~Punto2D(  )
```

Se tra i dati membri di una classe c'è un puntatore

Il compilatore automaticamente vi copierà nel puntatore l'indirizzo presente nel puntatore dell'oggetto originale…
Che succede quando il primo oggetto viene cancellato?
In questo caso

1. **Dovete definire e implementare un costruttore di copia**
2. **Dovete fornire e implementare un distruttore**
3. **Dovete ridefinire l'operatore di assegnazione operator=**
4. **Dovete ridefinire l'operatore di uguaglianza operator==**