-- class --

Verso gli oggetti

- struct + metodi = classi ? ... No !
  - nell' uso delle struct non ci sono praticamente regole: gli attributi sono visibili da tutti e modificabili da tutti
  - è utile limitare/controllare l'accesso agli attributi per implementare una vera *incapsulazione*.
  - Per questo nel C++ esistono 3 ambiti di visibilità delle variabili delle struct:

`public:`	dati/metodi accessibili a tutti
`private:`	dati/metodi accessibili solo al creatore della struct
`protected:`	come private più accesso per le "strutture ereditate" (!)

Principi di Parna rivisitati

chi implementa una ADT deve avere tutte le informazioni necessarie a realizzarne la struttura

e NULLA PIÙ

chi utilizza una ADT (client programmer) deve ricevere tutte le informazioni necessarie ad usarla

e NULLA PIÙ

Esempio di controllo di accesso

```
struct data {
private:
  int giorno;
  int mese;
  int anno;
public:
   void set (int, int, int);
   void get (int*, int*, int*);
   void next();
   void print();
};
```

controllo di accesso

```
void data::print() {
  cout << giorno << '/'
       << mese   << '/'
       << anno << '\n'
}

void backdate() {
  oggi.giorno--;
}
```

OK, come prima per le member function

ERRORE, backdate non è una member function e l'attributo giorno è **private** (non accessibile)

Friends

- Come consentire l'accesso alla parte private a struct/function esterne alla struct ?
- La direttiva FRIEND permette di dare accesso a singole function globali, singole function di altra struct, intere strutture (tutte le function di altra struct)

```
friend int FunGlobale(...);
friend void AStruct::AFun(...);
friend struct AnotherStruct;
```

- Anche per struct contenute in struct ...

Friend is not your friend ...

- L'abuso delle dichiarazioni Friend rende vana la protezione delle parti private di una struct
- Anche se spesso sembra più comodo usare soluzioni ad hoc dando accesso a singole function, si aumenta il grado di accoppiamento tra le varie parti del codice (ovvero la complessità)

Classi in C++

- classi: attributi + metodi + controllo di accesso + ...
- la differenza con le struct è che per default gli attributi sono private, mentre per le struct per default sono public

```
class data {
   int giorno; int mese; int anno;
public:
   void set (int, int, int);
   void get (int*, int*, int*);
   void next();
   void print();
};
```

Esempio 1

```
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  void initialize(int size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH_H
```

Esempio 2

```
#ifndef STACK2_H
#define STACK2_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
public:
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK2_H
```

Incapsulazione

- All’interno della interfaccia di un oggetto vengono nascosti tutti i dettagli tecnici relativi alla sua implementazione
- L’oggetto interagisce con l’esterno *solo* tramite i messaggi che riceve e manda
- Questo permette di lavorare a compartimenti stagni e rende il software (in teoria) più robusto e resistente ai cambiamenti

Pubblico/Privato

- Il C++ realizza questo concetto utilizzando l'idea di parte (o vista) *pubblica* e parte (o vista) *privata*
- La parte pubblica di un componente è formata da quelle parti (attributi e/o metodi) che gli altri componenti possono vedere e/o modificare
- La parte privata di un componente è formata da quelle parti (attributi e/o metodi) che possono essere usate e/o modificate solo dal componente medesimo

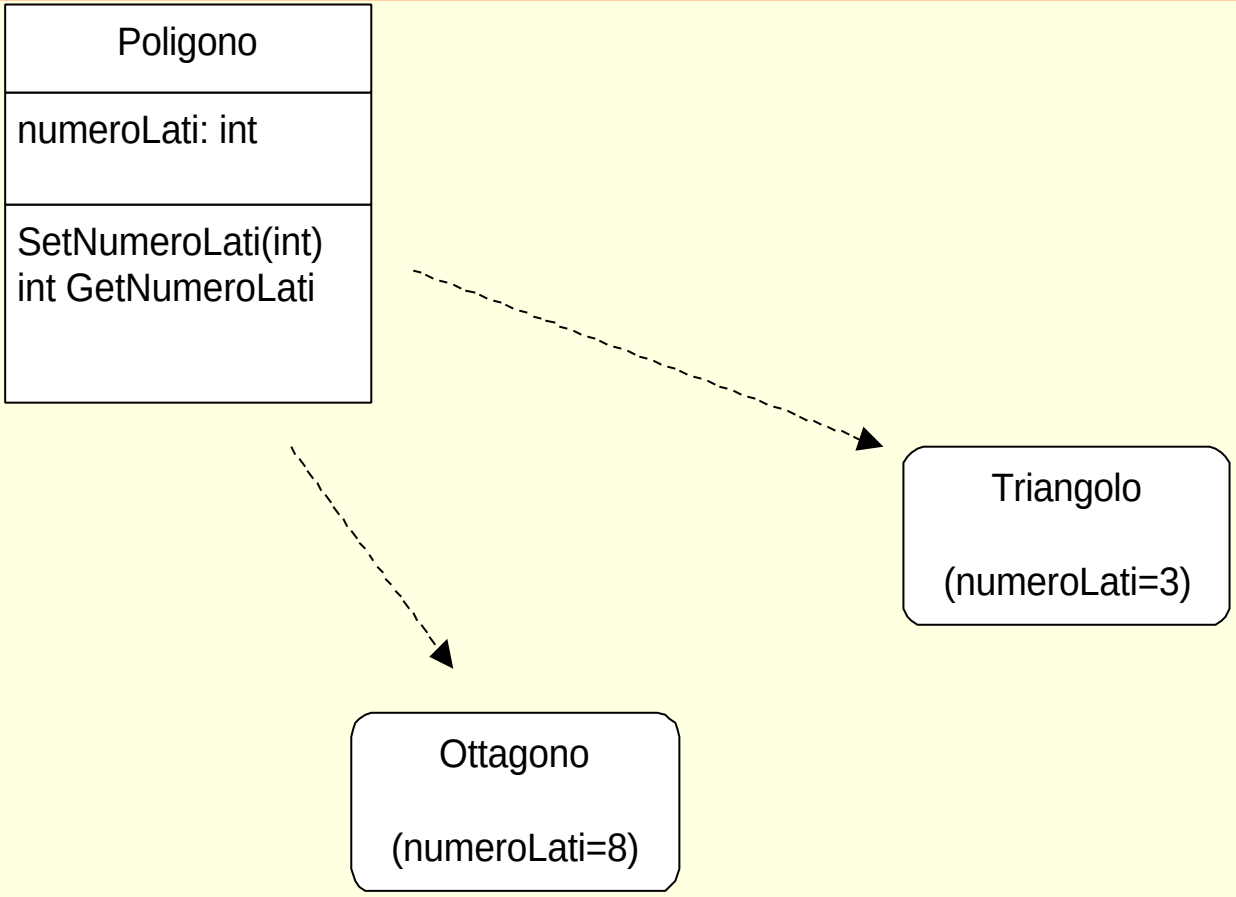
Metodi e messaggi

- Gli oggetti collaborano tra loro scambiandosi messaggi
- Ogni oggetto reagisce alla ricezione di un messaggio in un determinato modo, chiamato metodo
- I metodi sono uguali per tutti gli oggetti di una stessa classe - il comportamento di un oggetto dipende dalla sua classe

Messaggi / Funzioni

- Quando si trasmette un messaggio lo si trasmette sempre ad una istanza specifica, chiamata il *ricevitore*
- Sta al ricevitore decidere quale particolare metodo invocare a seguito della ricezione del messaggio
- Quando il metodo è stato invocato, viene passato anche il ricevitore come parametro implicito (esso è accessibile con la pseudovariabile puntatore *this* in C++)

Esempio


Poligono
numeroLati: int
SetNumeroLati(int)
int GetNumeroLati
Triangolo
(numeroLati=3)
Ottagono
(numeroLati=8)

Trasposizione in C++

Classe Poligono

Dichiarazione della classe con attributi e dichiarazione dei metodi (parte pubblica e privata) - corrisponde ad una dichiarazione di tipo

`Poligono.h`

Definizione dei metodi “concreti”

`Poligono.C`

File Poligono.h

```
#ifndef _Poligono_h_
#define _Poligono_h_
class Poligono
{
   private:
      int numeroLati;
   public:
      void SetNumeroLati(int);
      int GetNumeroLati();
};
#endif
```

Evita inclusioni multiple del file Poligono.h

Dichiarazione della classe con parte pubblica e privata

Attributi

Dichiarazione dei Metodi

File Poligono.C

```
#include "Poligono.h"
void Poligono::SetNumeroLati(int n)
{
  numeroLati=n;
}

int Poligono::GetNumeroLati()
{
  return numeroLati;
}
```

Implementazione dei metodi

Notare l'accesso agli attributi

Programma principale

```
#include <iostream>
#include "Poligono.h"
void main()
{
  Poligono ottagono,triangolo;

  ottagono.SetNumeroLati(8);
  triangolo.SetNumeroLati(3);

  cout << ottagono.GetNumeroLati();
  cout << triangolo.GetNumeroLati(); }
```

Costruzione di oggetti di classe Poligono

Invio messaggi a questi oggetti

Class Handle

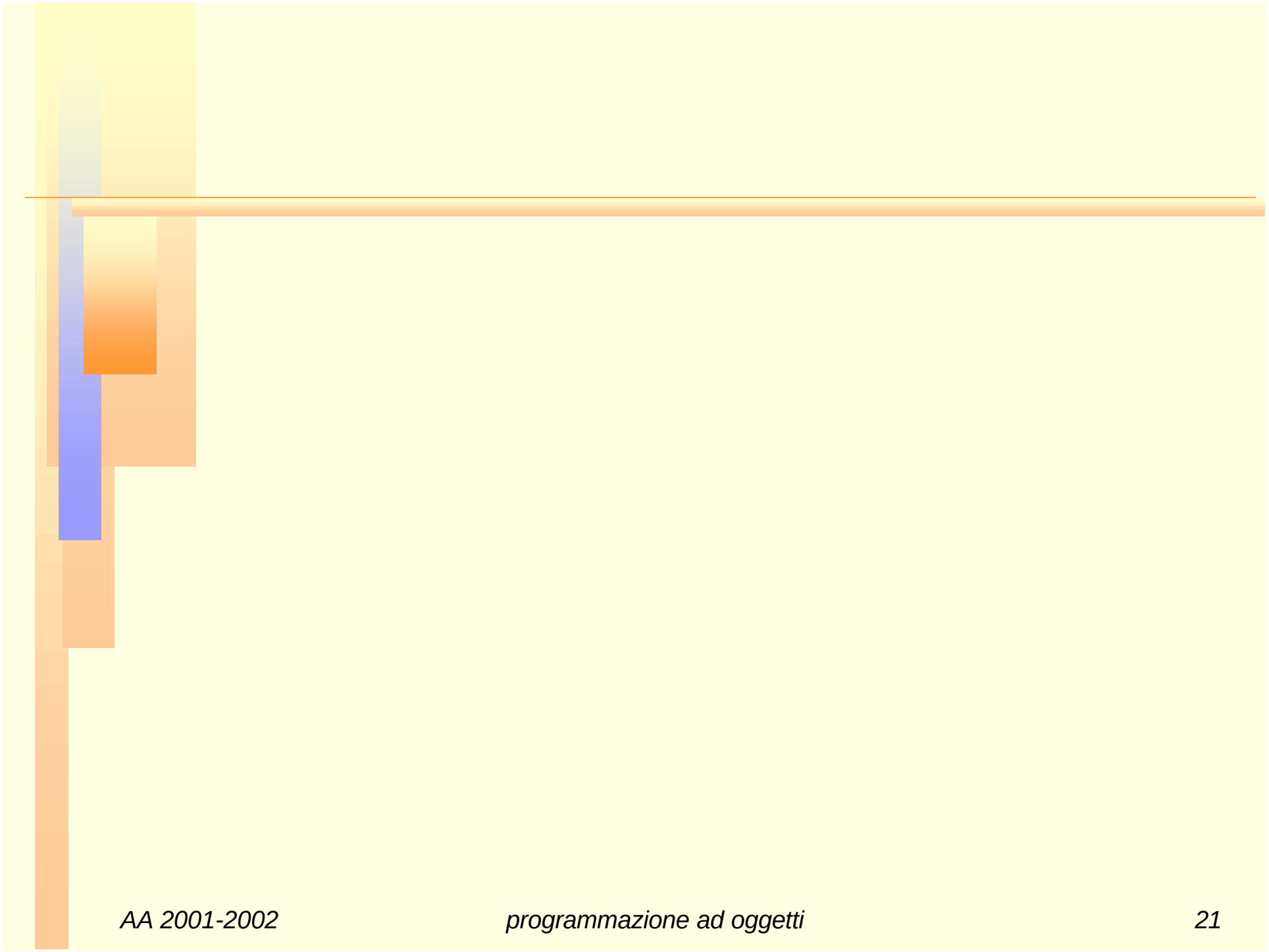
- Proteggere le dichiarazioni private
- Evitare ricompilazioni per modifiche

```
xyz.h
class xyz {
  struct hidden;
  hidden* veryprivate;
public:
  ...
}

xyz.cpp (distribuito solo compilato)
struct hidden {
   int mydata;
   ...
}
```

Dichiarazione incompleta

Dichiarazione completa



IOStream: input

- Per un maggior controllo sull'input si usano i metodi get() associati a *cin*.

legge un singolo carattere (qualsiasi)

```
cin.get (char c)
{char c; while(cin.get(c)) cout<<c; }
{char c; while((c=cin.get()) cout<<c; }
```

legge n caratteri nel vettore di caratteri che incomincia in p.

Il terzo argomento specifica un terminatore di lettura.

```
cin.get (char* p, int n, char ='\n');
cin.getline(char* p, int n, char ='\n');
{char buf[100]; cin.get(buf,100,'\n');}
```

IOStream: output

- Per un controllo maggiore sull'output si usano i metodi put() e write() associati a *cout*.
  - **`cout.put (char c)`**
    scrive un singolo carattere (qualsiasi)

```
{char c; while(cin.get(c)) cout.put(c); }
```

  - **`cout.write (char* p, int n)`**
    scrive in output n caratteri dal vettore di caratteri che incomincia in p.
    NB! non termina con il carattere di terminazione '\0' !

```
{char Title[]="uno due tre prova";
 cout.write(Title,5); cout<<'\n';
 cout.write(&Title[0],20);
}
```